

Projektarbeit: Diabetes

Kurs: Maschinelles Lernen (MaLe-AD23-HS24)

Autor/in: Christoph Reichlin

Datum: 18.12.2024

- [Website des Datensatzes](#)
-

Einführung

Problemstellung

Ziel ist es, mithilfe von medizinischen Daten vorherzusagen, ob ein Patient an Diabetes erkrankt ist oder nicht. Diese binäre Klassifikation soll mit Supervised Learning gelöst werden.

Daten

Der Datensatz umfasst:

- **Zeilen:** 2000
- **Spalten:** 9 (davon 1 Zielvariable)

Zielvariable

Die Zielvariable ist `Outcome`.

Features und Bedeutung

1. Pregnancies

- Anzahl der Schwangerschaften, die eine Person hatte.

2. Glucose

- Messung des Blutzuckerspiegel (in mg/dl) nach einer Glukosebelastung.

3. Blood Pressure

- Blutdruck (in mmHg) diastolisch gemessen.

4. Skin Thickness

- Hautfaltendicke (in mm), gemessen an der Trizeps-Stelle.

5. Insulin

- Serum-Insulinspiegel (in $\mu\text{U/ml}$).

6. BMI

- Body Mass Index (in kg/m^2).

7. Diabetes Pedigree Function

- Ein berechneter Wert, der die genetische Wahrscheinlichkeit von Diabetes anhand der Familienanamnese beschreibt.

8. Age

- Alter der Person (in Jahren).

9. Zielvariable

- 0 = Nein : Kein Diabetes
- 1 = Ja : Diabetes vorhanden

10. Insulin Missing

- 1 = Insulin Messwert nicht vorhanden
- 0 = Insulin Messwert vorhanden

Ressourcen & Einflüsse

- [A model for early prediction of diabetes](#)
- [An Enhanced Machine Learning Framework for Type 2 Diabetes Classification Using Imbalanced Data with Missing Values](#)
- [Chatgpt](#)

Anmerkung: Bei den ersten Aufgaben von Modellieren, Validieren und Testen wurde verstärkt auf KI-Tools zurückgegriffen.

Setup

Anleitung: In diesem Abschnitt geht es darum, das Jupyter Notebook zu konfigurieren.

```
In [43]: # Basic imports
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
import pandas as pd
from sklearn.impute import SimpleImputer
from sklearn.ensemble import RandomForestClassifier
from sklearn.impute import KNNImputer
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import GridSearchCV, train_test_split, cross_val_score
from sklearn.metrics import roc_curve, auc
from sklearn.ensemble import RandomForestClassifier
from sklearn.linear_model import LogisticRegression
```

```

from sklearn.metrics import classification_report, accuracy_score, precision_score
from sklearn.svm import SVR
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay

# Enable vectorized graphics
%config InlineBackend.figure_formats = ["svg"]

# Setup plotting
PALETTE = [ (0.341, 0.648, 0.962, 1.0),
            (0.990, 0.476, 0.494, 1.0),
            (0.281, 0.749, 0.463, 1.0),
            (0.629, 0.802, 0.978, 1.0),
            (0.994, 0.705, 0.715, 1.0),
            (0.595, 0.858, 0.698, 1.0),
            (0.876, 0.934, 0.992, 1.0),
            (0.998, 0.901, 0.905, 1.0),
            (0.865, 0.952, 0.899, 1.0) ]

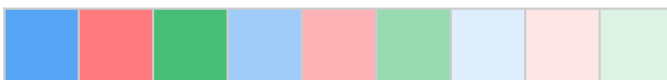
# For more color palettes, see here:
# https://seaborn.pydata.org/tutorial/color_palettes.html
# https://matplotlib.org/stable/users/explain/colors/colormaps.html
#PALETTE = sns.color_palette("husl", 8)
#PALETTE = sns.color_palette("viridis", 10)

print("Our color palette:")
sns.palplot(PALETTE, size=0.5)

sns.set_style("whitegrid")
plt.rcParams["axes.prop_cycle"] = plt.cycler(color=PALETTE)
plt.rcParams["figure.dpi"] = 300 # High-res figures (DPI)
plt.rcParams["pdf.fonttype"] = 42 # Editable text in PDF

```

Our color palette:



Präprozessierung

Anleitung: In diesem Abschnitt werden die Daten geladen und aufbereitet.

```

In [44]: # Code to load and preprocess the data to the variable diabetes
diabetes = pd.read_csv("diabetes-dataset.csv")

```

```

In [45]: #show the first 10 rows of the dataset to get an idea of the data
diabetes.head(10)

```

Out[45]:

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigree
0	2	138	62	35	0	33.6	
1	0	84	82	31	125	38.2	
2	0	145	0	0	0	44.2	
3	0	135	68	42	250	42.3	
4	1	139	62	41	480	40.7	
5	0	173	78	32	265	46.5	
6	4	99	72	17	0	25.6	
7	8	194	80	0	0	26.1	
8	2	83	65	28	66	36.8	
9	2	89	90	30	0	33.5	

Explorative Datenanalyse

Anleitung: Hier untersucht ihr eure Daten ein erstes Mal, visualisiert sie und versucht, Muster zu erkennen. Zeigt hier, dass ihr neugierig seid und euch mit den Daten auseinandersetzt.

In [46]:

```
# describe the dataset
diabetes.describe()
```

Out[46]:

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI
count	2000.000000	2000.000000	2000.000000	2000.000000	2000.000000	2000.000000
mean	3.703500	121.182500	69.145500	20.935000	80.254000	32.193000
std	3.306063	32.068636	19.188315	16.103243	111.180534	8.149900
min	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
25%	1.000000	99.000000	63.500000	0.000000	0.000000	27.375000
50%	3.000000	117.000000	72.000000	23.000000	40.000000	32.300000
75%	6.000000	141.000000	80.000000	32.000000	130.000000	36.800000
max	17.000000	199.000000	122.000000	110.000000	744.000000	80.600000

- Hier wird der Datensatz mit `df.describe()` das erste mal untersucht, um Werte wie die Anzahl, Mittelwert usw. von allen Spalten zu sehen. Zudem sieht man gleich

ob etwas Unstimmig ist, wie die 0 Stellen in den Spalten Glucose , Blood Pressure , Skin Thickness , Insulin und BMI .

```
In [47]: #info about the dataset
diabetes.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2000 entries, 0 to 1999
Data columns (total 9 columns):
#   Column                Non-Null Count  Dtype
---  -
0   Pregnancies            2000 non-null   int64
1   Glucose                2000 non-null   int64
2   BloodPressure          2000 non-null   int64
3   SkinThickness          2000 non-null   int64
4   Insulin                2000 non-null   int64
5   BMI                   2000 non-null   float64
6   DiabetesPedigreeFunction 2000 non-null   float64
7   Age                   2000 non-null   int64
8   Outcome                2000 non-null   int64
dtypes: float64(2), int64(7)
memory usage: 140.8 KB
```

- Mit `df.info` erkennt man die Datentypen der Spalten. Gut zu erkennen ist, dass nur `integer` und `floats` vorkommen.

```
In [48]: # Count duplicates in the dataset and store the result in duplicate_count
duplicate_count = diabetes.duplicated().sum()

# Print the number of duplicates in the dataset
print(f"Anzahl der doppelten Zeilen im Datensatz: {duplicate_count}")
```

Anzahl der doppelten Zeilen im Datensatz: 1256

- Mit `df.duplicated().sum()` werden alle Zeilen, welche Duplikate enthalten aufsummiert, um zu erkennen, wie viele Duplikate es gibt. Das sind insgesamt 1256 Zeilen von 2000 Zeilen.

```
In [49]: # Proportion of missing values in the dataset
row_counts = diabetes.value_counts()
print(row_counts[row_counts > 1]) # Show rows with more than one occurrence
```

Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigree
Function	Age	Outcome				
2		81	72	15	76	30.1 0.547
25	0	6				
		83	65	28	66	36.8 0.629
24	0	5				
6		154	74	32	193	29.3 0.839
39	0	5				
3		80	0	0	0.0	0.174
22	0	5				
4		125	70	18	122	28.9 1.144
45	1	5				
..						
5		96	74	18	67	33.6 0.997
43	0	2				
		95	72	33	0	37.7 0.370
27	0	2				
1		97	70	40	0	38.1 0.218
30	0	2				
5		88	66	21	23	24.4 0.342
30	0	2				
		109	62	41	129	35.8 0.514
25	1	2				

Name: count, Length: 730, dtype: int64

- Hier wird mit `df.value_counts()` die Häufigkeit von Zeilen angegeben, welche öfters, als 1 mal vorkommen. Wie man schön sieht kommen gewisse zwischen 2 bis und mit 6 mal vor.

```
In [50]: # Remove duplicates from the dataset
diabetes_cleaned = diabetes.drop_duplicates()

# Count the number of rows after removing duplicates
print(f"Anzahl der Zeilen nach dem Entfernen der Duplikate: {diabetes_cleaned.shape[0]}")
```

Anzahl der Zeilen nach dem Entfernen der Duplikate: 744

- Mit `df.drop_duplicates()` werden alle Duplikate entfernt. Am Schluss sind noch 744 Zeilen vorhanden nach dem bereinigen. Dieses wird in der neuen Variable `diabetes_cleaned` gespeichert.

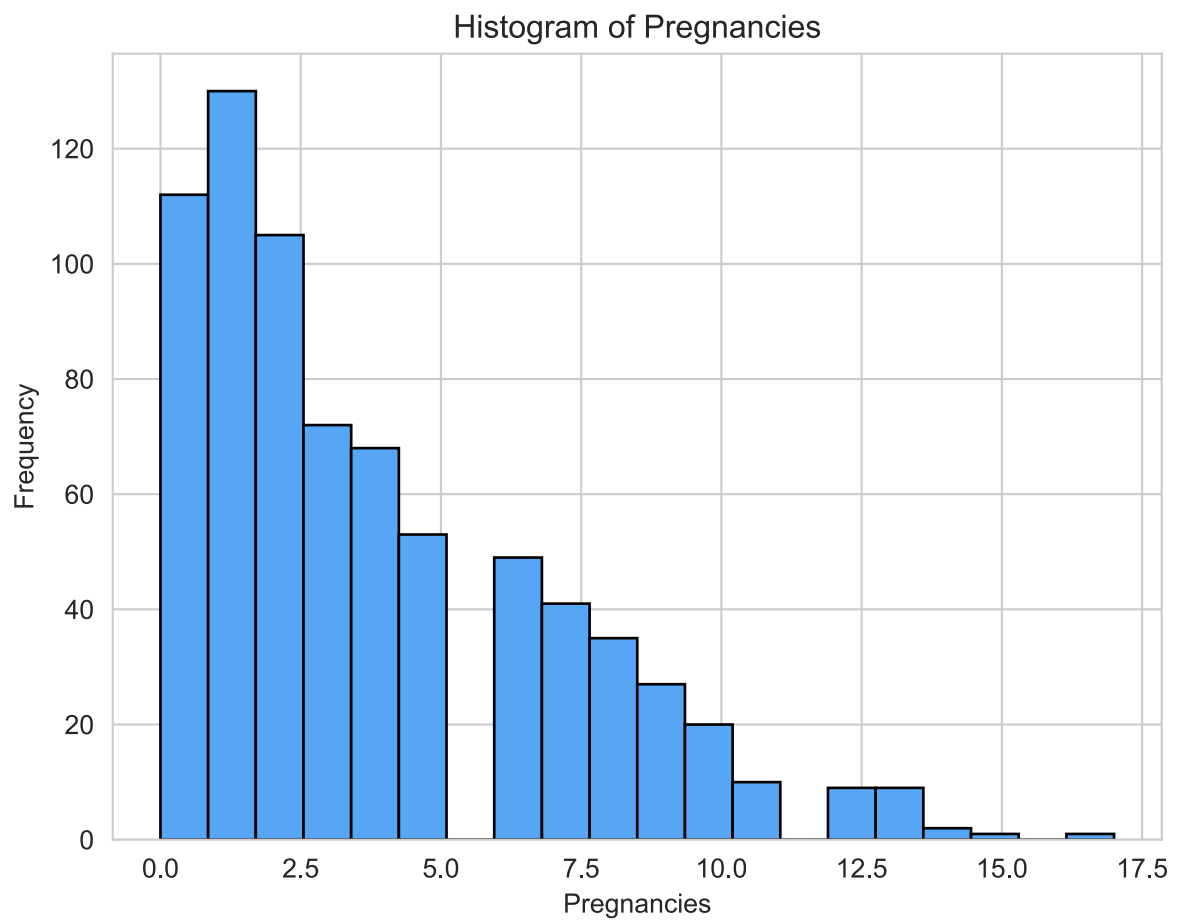
```
In [51]: # Loop through all numerical columns in the dataset
for column in diabetes_cleaned.select_dtypes(include='number').columns:
    plt.figure(figsize=(12, 5)) # Set figure size

    # Create a histogram for the column
    plt.subplot(1, 2, 1) # First subplot
    diabetes_cleaned[column].hist(bins=20, edgecolor='black')
    plt.title(f'Histogram of {column}')
    plt.xlabel(column)
    plt.ylabel('Frequency')

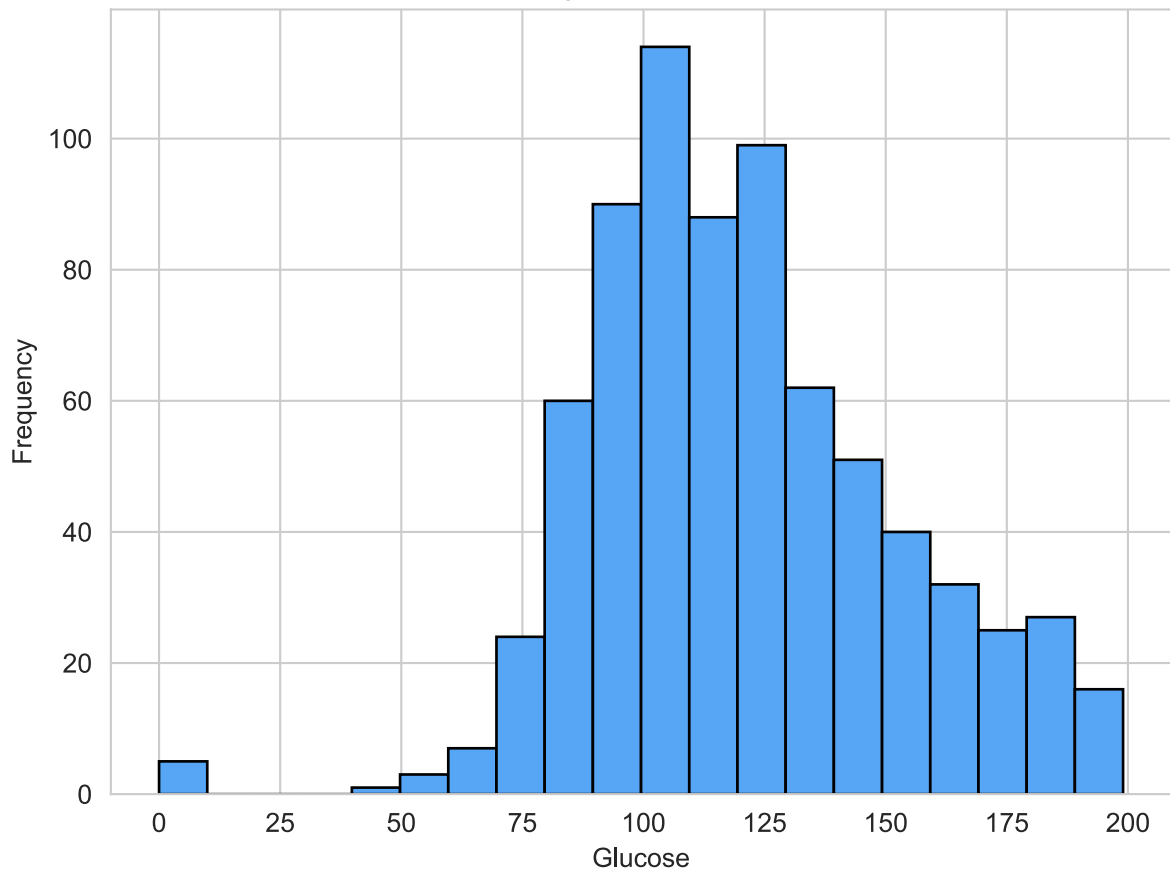
    # Create a boxplot for the column
    plt.subplot(1, 2, 2) # Second subplot
    sns.boxplot(x=diabetes[column])
    plt.title(f'Boxplot of {column}')
```

```
plt.xlabel(column)
```

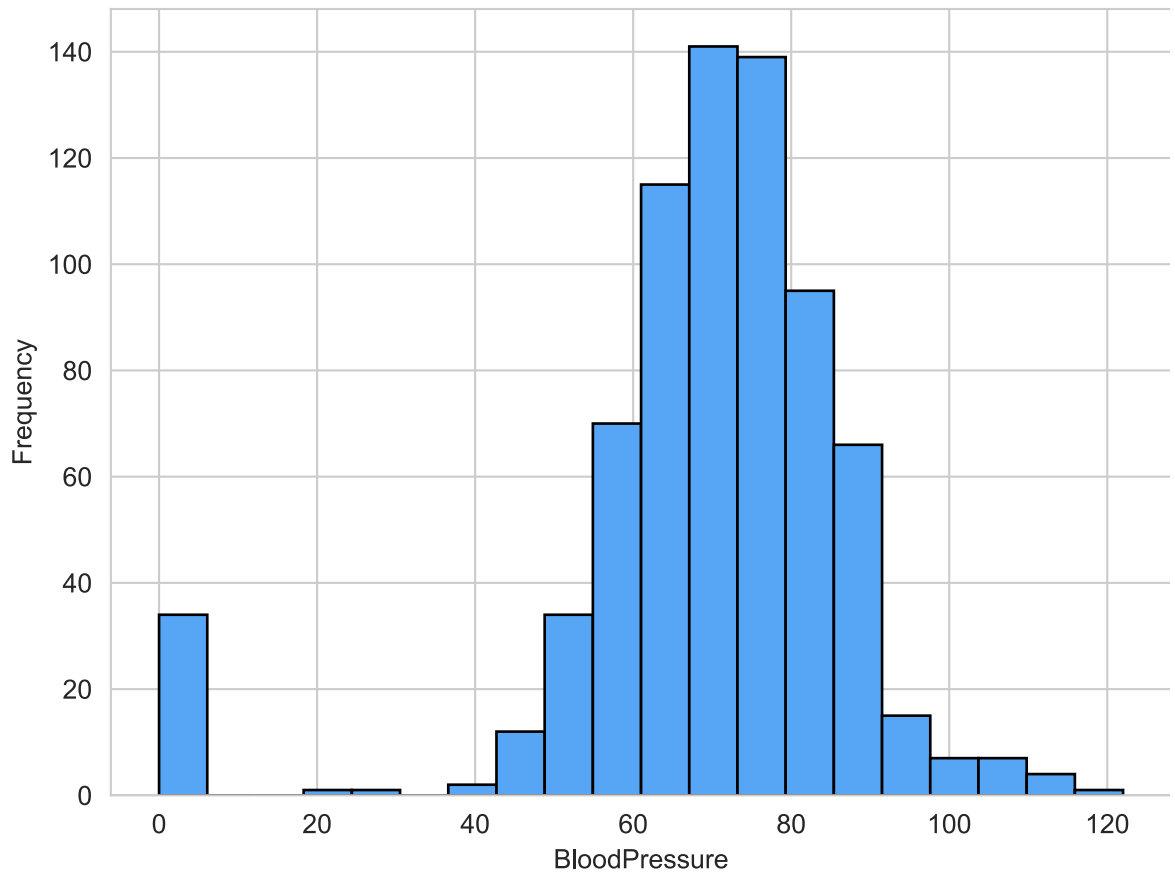
```
plt.tight_layout() # Adjust layout to prevent overlap  
plt.show()
```



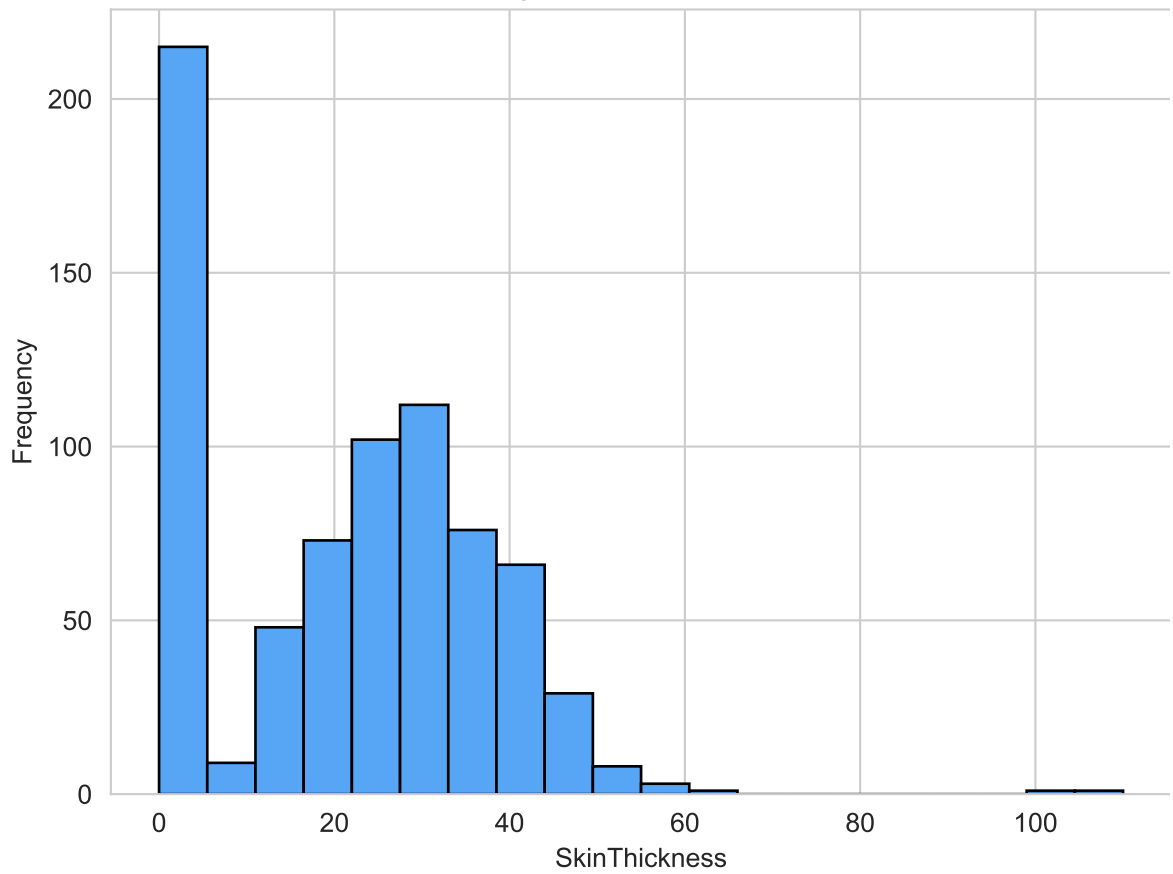
Histogram of Glucose



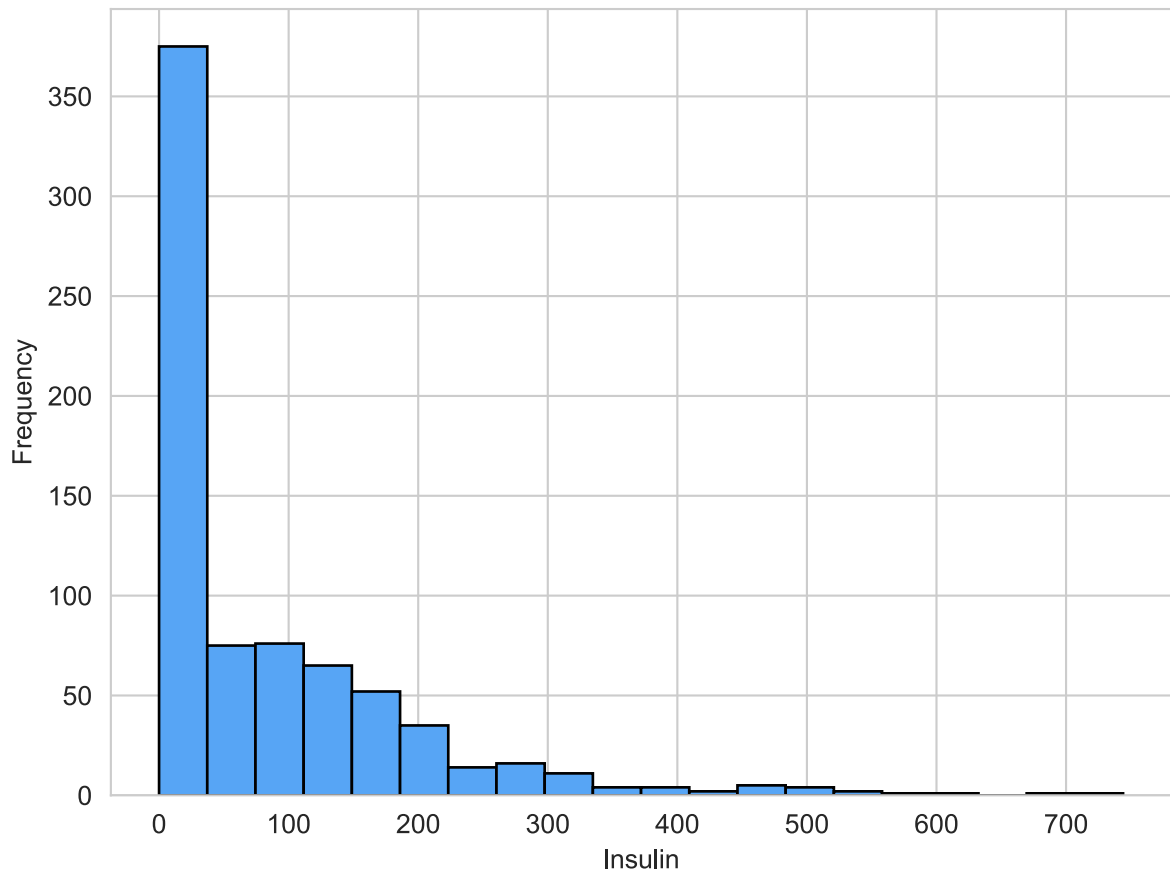
Histogram of BloodPressure



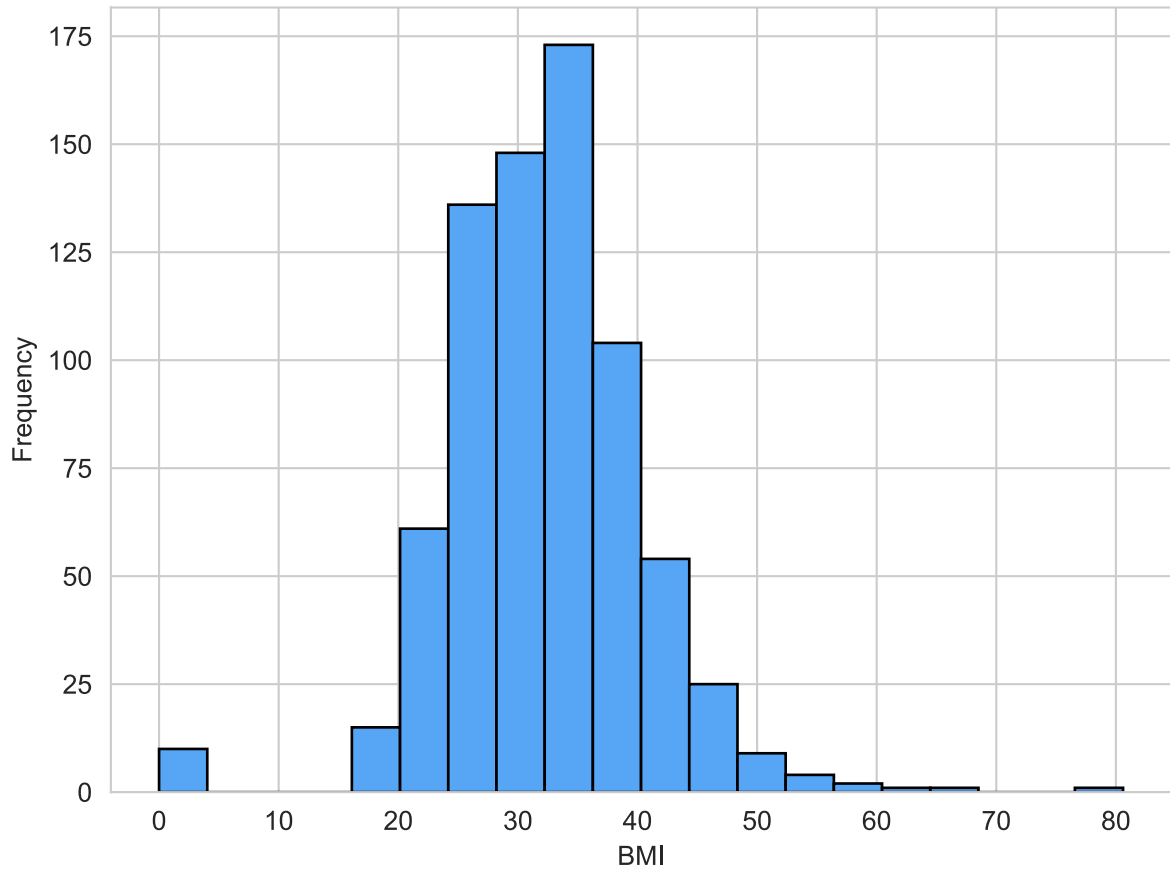
Histogram of SkinThickness



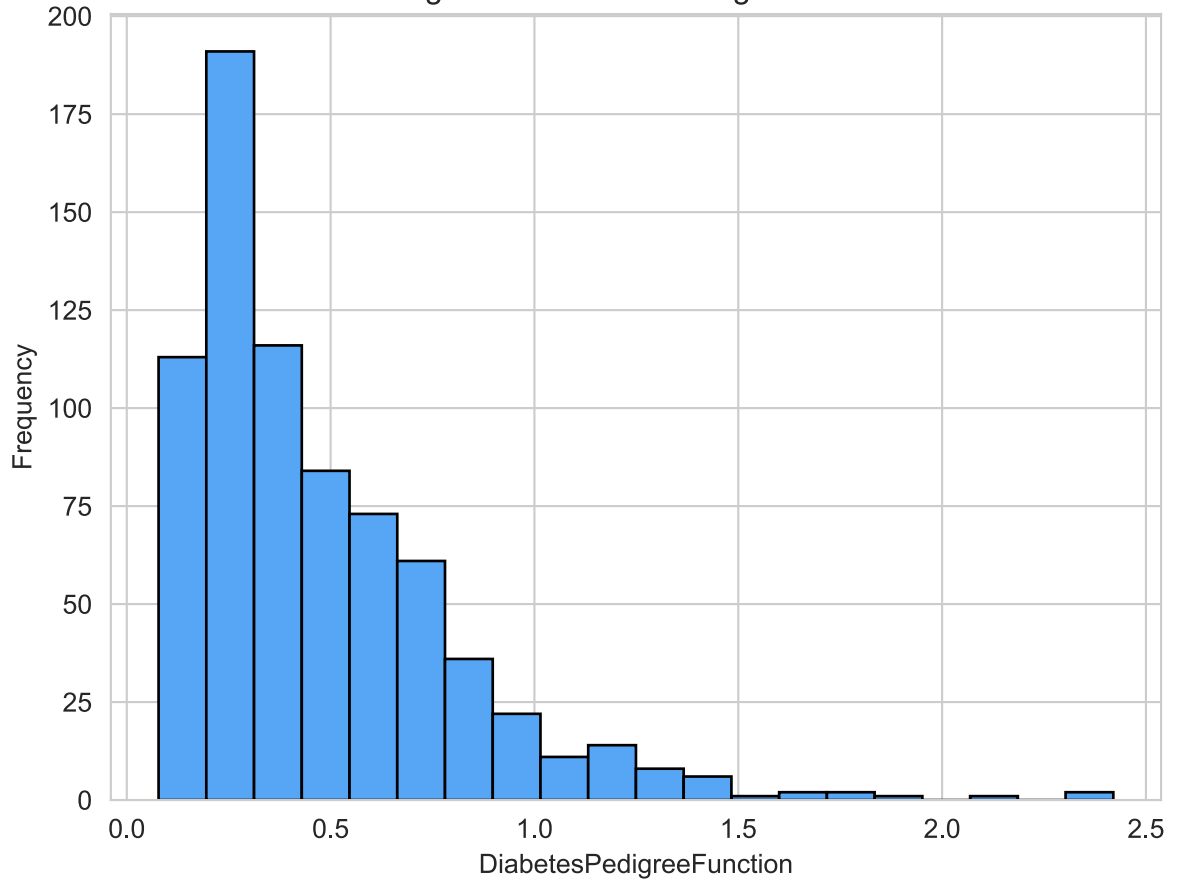
Histogram of Insulin



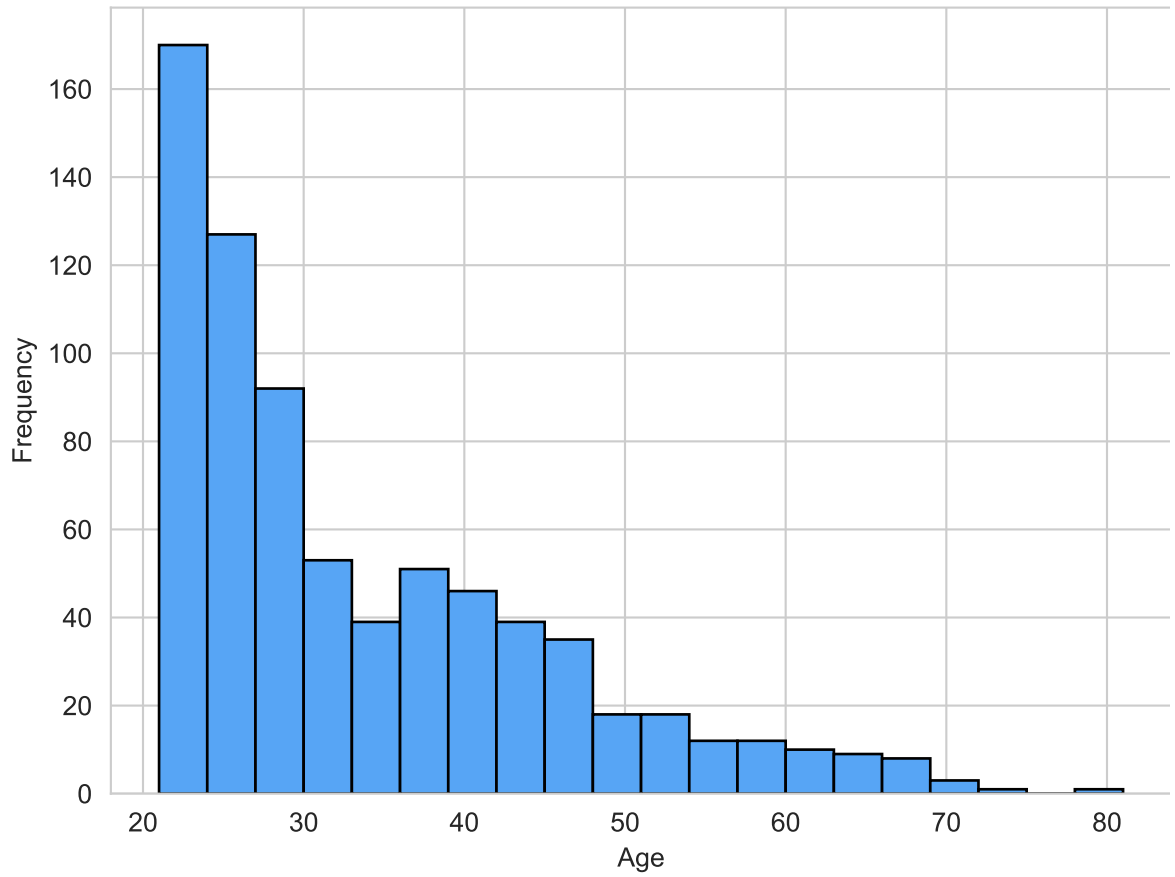
Histogram of BMI



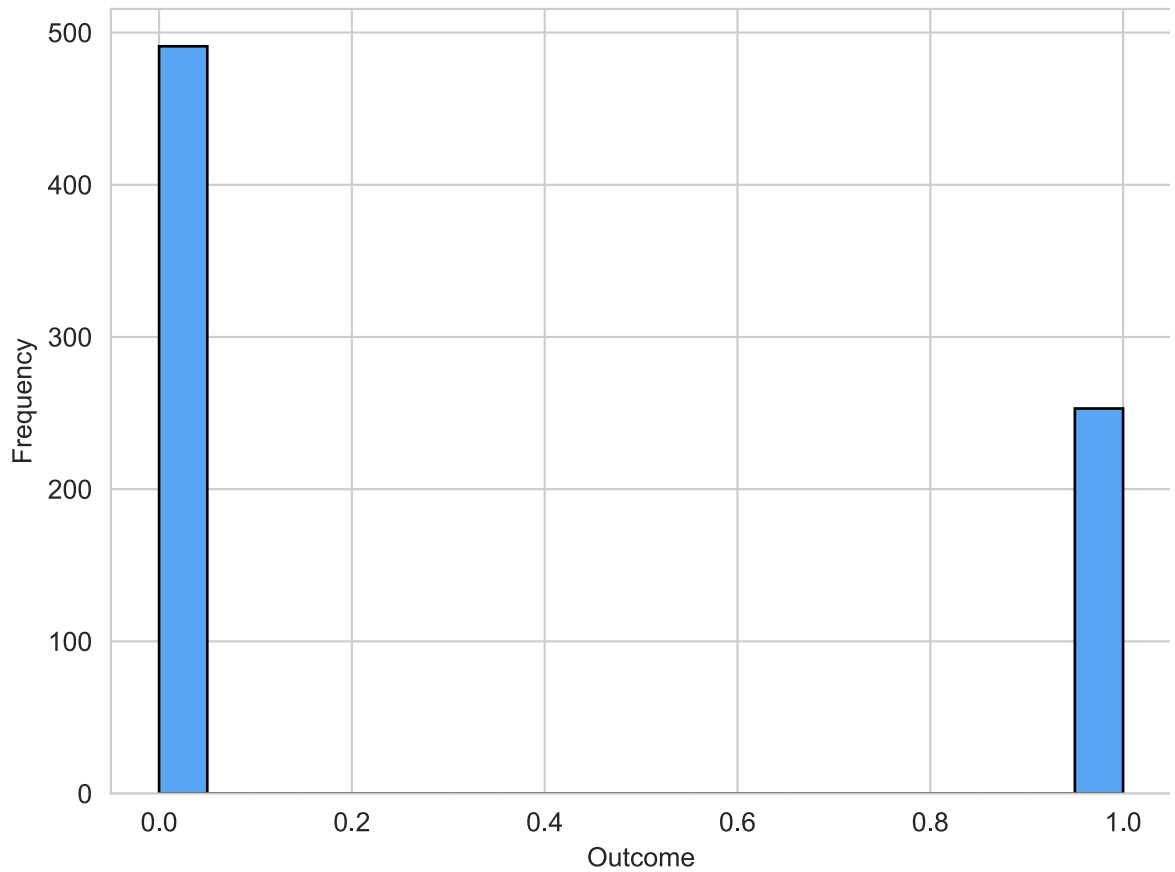
Histogram of DiabetesPedigreeFunction



Histogram of Age



Histogram of Outcome



Output

- Der Output bei dieser Aufgabe wird mit einer `for` Schleife generiert. Das sind die Histogramme und Boxplots der einzelnen Kategorien.

Histogramme

1. `Pregnancies`
 - rechtsschief Verteilung
2. `Glucose`
 - annähernde Normalverteilung
3. `Blood Pressure`
 - annähernde Normalverteilung
4. `Skin Thickness`
 - rechtsschief Verteilung
5. `Insulin`
 - rechtsschief Verteilung
6. `BMI`
 - annähernde Normalverteilung
7. `Diabetes Pedigree Function`
 - rechtsschief Verteilung
8. `Age`
 - rechtsschief Verteilung
9. `Outcome`
 - binäre Verteilung. `0 = kein Diabetes` kommt weitaus häufiger vor als `1 = Diabetes`. Somit habe ich es hier mit dem `imbalanced data` Problem zu tun.
 - Anmerkung: Die Anzahl der `0` Werte sind jeweils auf der linken Seite der Histogramme gut zu erkennen.

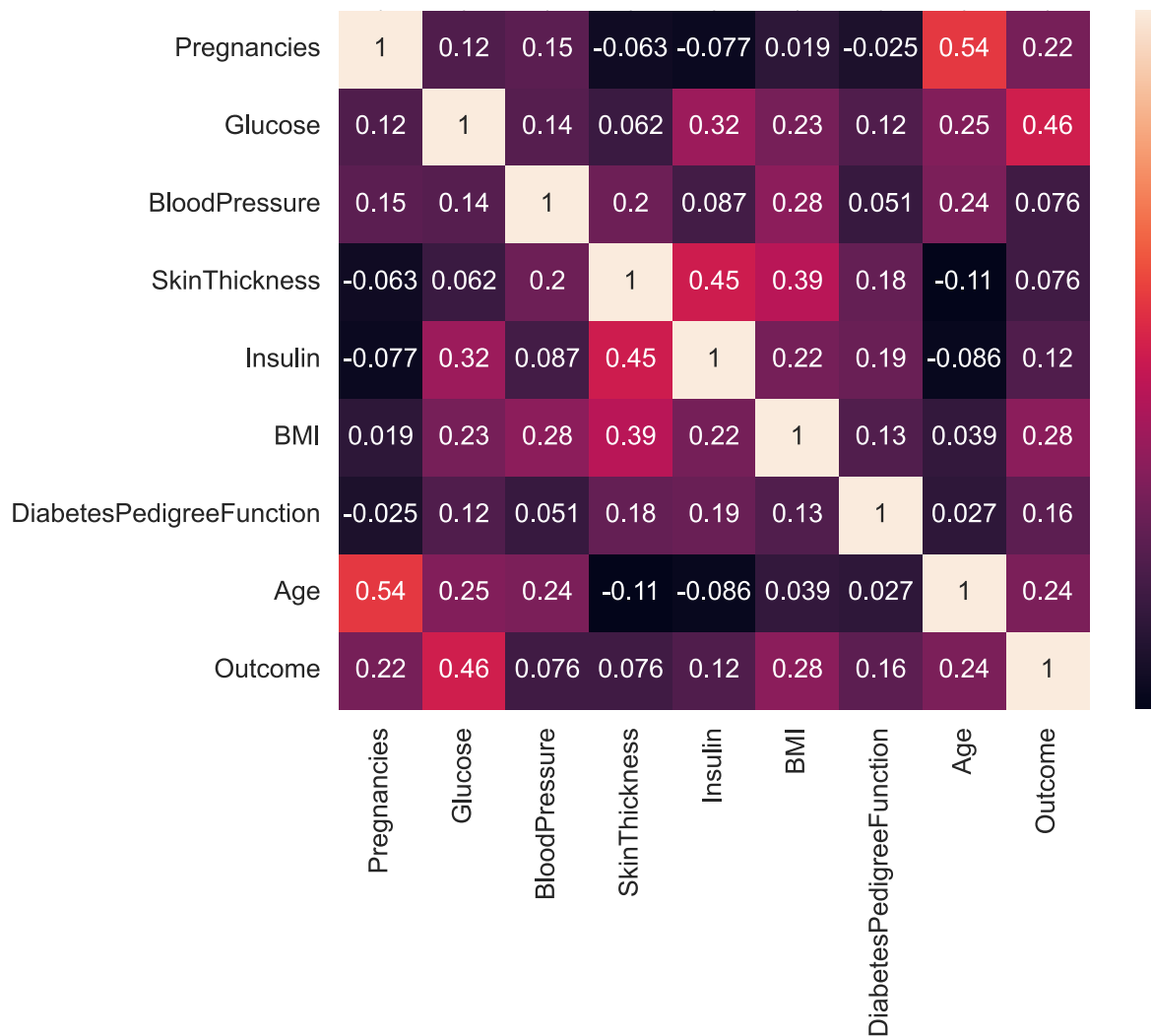
Boxplots

- Die 9 Boxplots zeigen die Verteilung, Streuung und potenzielle Ausreißer der Gesundheitsdaten für jede Kategorie.

```
In [52]: #corr() function to find the pairwise correlation of all columns in the dataframe
diabetes_cleaned.corr()
```

```
#heatmap to visualize the correlation
sns.heatmap(diabetes.corr(), annot=True)
```

Out[52]: <Axes: >



- Mit dieser **Korrelationsmatrix** werden lineare Zusammenhänge dargestellt. Werte nahe **-1** zeigen eine starke negative Korrelation und Werte nahe **1** zeigen eine starke positive Korrelation. Zudem sind Werte nahe **0** ein Hinweis für keinen linearen Zusammenhang zwischen den Kategorien.

```
In [53]: # Count how often 0 appears in each column of the dataset
zero_counts = (diabetes_cleaned == 0).sum()

# Print the counts of 0 values for each column in the dataset
print("Count of 0 values in each column:")
print(zero_counts)
```

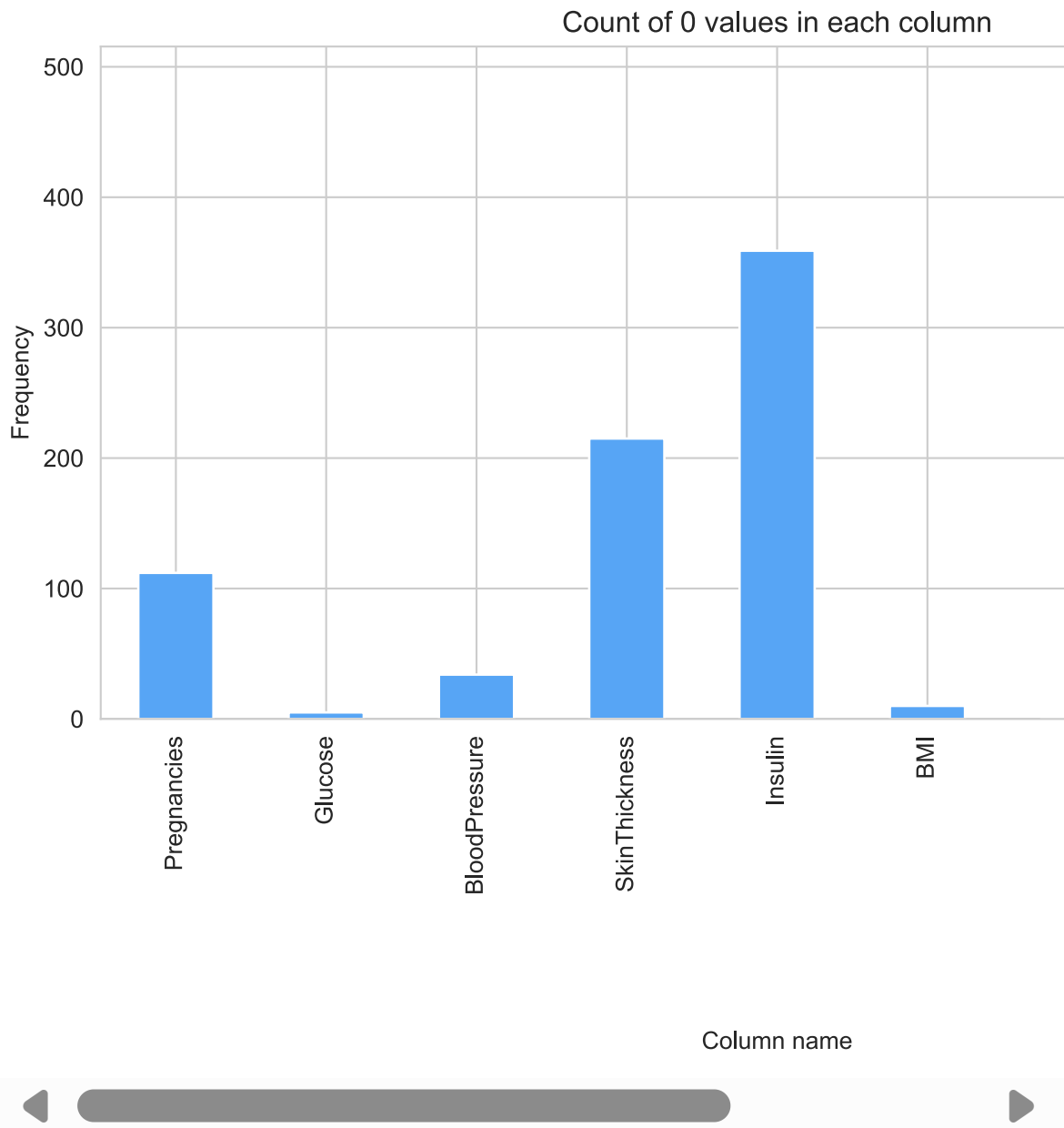
Count of 0 values in each column:

Pregnancies	112
Glucose	5
BloodPressure	34
SkinThickness	215
Insulin	359
BMI	10
DiabetesPedigreeFunction	0
Age	0
Outcome	491

dtype: int64

- Mit `(df == 0).sum()` zähle ich, wie oft die 0 vorkommt. Diese wird dann mit `print` ausgegeben. Hier sieht man das Insulin und Skin Thickness sehr viele 0 Werte haben.

```
In [54]: #visualize zero counts in each column using a bar plot
plt.figure(figsize=(10, 5))
zero_counts.plot(kind='bar')
plt.title("Count of 0 values in each column")
plt.ylabel("Frequency")
plt.xlabel("Column name")
plt.show()
```



- Die Häufigkeit von 0 Werten wird mit einem Bar Plot dargestellt.

```
In [55]: #number of unique values in each column of the dataset
unique_counts = diabetes_cleaned.nunique()

# Print the number of unique values for each column in the dataset
print("Number of unique values in each column:")
print(unique_counts)
```

Number of unique values in each column:

Pregnancies	17
Glucose	136
BloodPressure	47
SkinThickness	53
Insulin	182
BMI	247
DiabetesPedigreeFunction	505
Age	52
Outcome	2

dtype: int64

- Mit `df.nunique()` werden die Anzahl einzigartiger Werte jeder Spalte angegeben. Die wenigsten hat selbstverständlich `Outcome` da diese Kategorie binär ist und bei `Diabetes Pedigree Function` haben wir die meisten mit `505` verschiedenen Werten

Feature Engineering und Dimensionalitätsreduktion

***Anleitung:** Beim Feature Engineering versucht ihr aus den verfügbaren Daten nützliche neue Features zu generieren. Falls ihr bereits viele Prädiktoren habt, versucht ihr die Dimensionalität des Problems mittels eines geeigneten Verfahrens zu reduzieren. Je nach Datensatz und Problem fällt dieser Abschnitt länger oder kürzer aus.*

```
In [56]: #copy the dataset
diabetes_new = diabetes_cleaned.copy()
```

- Mit `df.copy()` wird eine exakte Kopie des Dataframes erstellt, um das Original nicht zu beeinflussen.

```
In [57]: # new column to indicate missing values for Insulin
diabetes_new['Insulin_Missing'] = (diabetes_new['Insulin'] == 0).astype(int)

# Count the number of 1s and 0s in the new column
diabetes_new['Insulin_Missing'].value_counts()
```

```
Out[57]: Insulin_Missing
0      385
1      359
Name: count, dtype: int64
```

- Hier wird eine neue Kolonne erstellt für fehlende `Insulin` Werte, welche binär kategorisiert sind mit `1 = nicht vorhanden` und `0 = vorhanden`. Danach werden die Werte mit `df.value_counts()` gezählt und ausgegeben.

```
In [58]: # impute missing values with the median
imputer = SimpleImputer(missing_values=0, strategy='median')
columns_to_impute = ['Glucose', 'BloodPressure', 'BMI']

# only impute the columns with missing values
diabetes_new[columns_to_impute] = imputer.fit_transform(diabetes_new[columns_to_impute])

#head of data column
diabetes_new.head(5)
```


Out[58]:

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigree
0	2	138.0	62.0	35	0	33.6	
1	0	84.0	82.0	31	125	38.2	
2	0	145.0	72.0	0	0	44.2	
3	0	135.0	68.0	42	250	42.3	
4	1	139.0	62.0	41	480	40.7	



- Bei diesem Code Snippet wird mit dem `SimpleImputer` die fehlenden `0` Werte mit dem `Median` aufgefüllt für die Kolonnen Glucose, Blood Pressure und BMI. Anschliessend mit `imputer.fit_transform()` und als letztes mit `df.head(5)`, die ersten 5 Zeilen ausgegeben.

```
In [59]: # Create the KNN imputer
knn_imputer = KNNImputer(n_neighbors=5,missing_values=0)

# Columns to impute
columns_to_impute = ['Insulin', 'SkinThickness']

# Impute the missing values
diabetes_new[columns_to_impute] = knn_imputer.fit_transform(diabetes_new[columns_to_impute])

# Count the number of 0s in the Insulin column
print("Number of 0s in the Insulin column after imputation:", (diabetes_new['Insulin'] == 0).sum())

# Count the number of 0s in the SkinThickness column
print("Number of 0s in the SkinThickness column after imputation:", (diabetes_new['SkinThickness'] == 0).sum())


diabetes_new.head(10)
```

Number of 0s in the Insulin column after imputation: 0

Number of 0s in the SkinThickness column after imputation: 0

Out[59]:

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedig
0	2	138.0	62.0	35.000000	213.600000	33.6	
1	0	84.0	82.0	31.000000	125.000000	38.2	
2	0	145.0	72.0	29.406427	153.698701	44.2	
3	0	135.0	68.0	42.000000	250.000000	42.3	
4	1	139.0	62.0	41.000000	480.000000	40.7	
5	0	173.0	78.0	32.000000	265.000000	46.5	
6	4	99.0	72.0	17.000000	110.600000	25.6	
7	8	194.0	80.0	29.406427	153.698701	26.1	
8	2	83.0	65.0	28.000000	66.000000	36.8	
9	2	89.0	90.0	30.000000	172.400000	33.5	



- Das Ziel ist es die 0 Stellen der Kolonnen Insulin und Skin Thickness mit dem KNN Imputer aufzufüllen. Die `n_neighbors = 5` wurde hier auf default eingestellt, wie in der KNN Imputer Bibliothek von [Scikit](#).
- Danach mit `fit_transform` angewendet und kontrolliert wie viele 0 Werte noch vorhanden sind. Am Schluss wieder mit `df.head(10)` die 10 ersten Zeilen ausgegeben.

```
In [60]: # Count how often 0 appears in each column
zero_counts_new = (diabetes_new == 0).sum()

# Print the counts of 0 values for each column
print("Count of 0 values in each column:")
print(zero_counts_new)
```

```
Count of 0 values in each column:
Pregnancies          112
Glucose              0
BloodPressure        0
SkinThickness        0
Insulin              0
BMI                  0
DiabetesPedigreeFunction  0
Age                  0
Outcome              491
Insulin_Missing      385
dtype: int64
```

- Nochmals die 0 Werte ausgeben zur letzten Kontrolle. Wie man schön sieht gibt es welche in den Kategorien `Pregnancies`, `Outcome` und `Insulin_Missing`, da es binäre Kategorien sind und 0 Schwangerschaften selbstverständlich vorkommen.

```
In [61]: # Create a random forest classifier
rf = RandomForestClassifier(n_estimators=100, random_state=42)

# Split the data into features and target
X = diabetes_new.drop(columns=['Outcome'])
y = diabetes_new['Outcome']

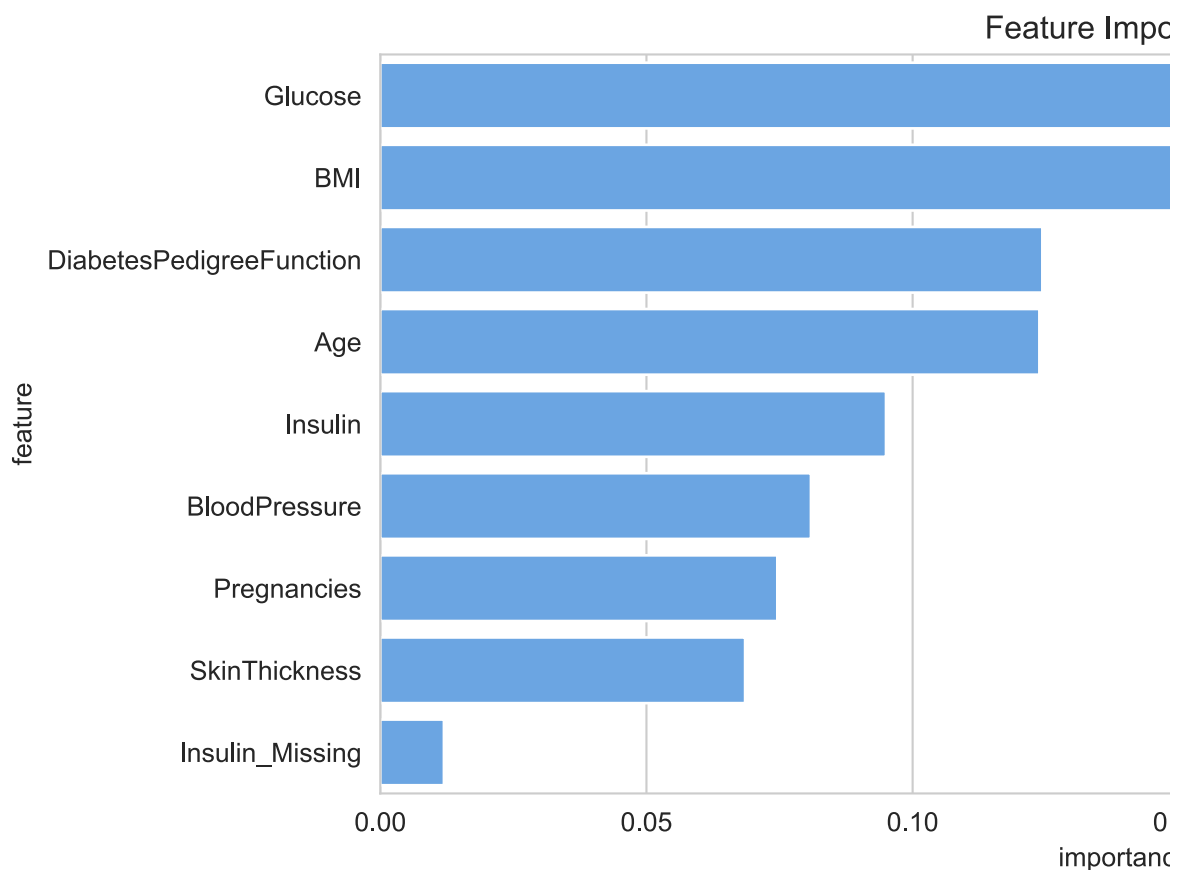
# Fit the classifier
rf.fit(X, y)

# Get the feature importances
importances = rf.feature_importances_

# Create a DataFrame
feature_importances = pd.DataFrame({'feature': X.columns, 'importance': importances})

# Sort values by importance
feature_importances = feature_importances.sort_values('importance', ascending=False)

# Plot a bar chart
plt.figure(figsize=(10, 5))
sns.barplot(x='importance', y='feature', data=feature_importances)
plt.title("Feature Importance")
plt.show()
```



- Mit `Random Forest Classifier` wird ein Modell trainiert, um zu sehen, wie wichtig die Merkmale für die `Outcome` Zielvariable sind und als Diagramm dargestellt.

Modellieren, Trainieren und Validieren

Anleitung: Es folgt der für diese Projektarbeit zentrale Abschnitt. Hier trainiert Datenmodelle, und validiert diese. Folgende Punkte sind zu beachten:

- Es sollen mindestens zwei verschiedene Datenmodelle trainiert und verglichen werden.
- Die Hyperparameter sollen mittels Kreuzvalidierung ermittelt werden.
- Die Vorhersagegenauigkeit der Modelle müssen mit einem separaten Testdatensatz abgeschätzt werden.
- Die Resultate der Trainings- und Testphasen sollen visualisiert werden (z.B. mittels ROC-Kurve oder Residuen-Plots).
- Bonuspunkte gibt es, falls
 - der Generalisierungsfehler des Modells robust ermittelt wird (durch wiederholtes Validieren, so dass die Kennzahlen für Vorhersagegenauigkeit als " $\mu \pm \sigma$ " angegeben werden kann).
 - die Vorhersagefehler der Modelle auf Muster untersucht werden.
 - falls eine Standardmodellierung basierend auf Beobachtungen verfeinert und weiterentwickelt wird.

Wir empfehlen, die verschiedenen Modelle in `scikit-learn` als `Pipelines` aufzubauen.

```
In [62]: # Split the data
X = diabetes_new.drop(columns=['Outcome']) # Features
y = diabetes_new['Outcome'] # Target variable
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_st

# Define the models
models = {
    'LogisticRegression': {
        'model': LogisticRegression(max_iter=1000, random_state=42), # Logistic R
        'params': {
            'model__C': [0.01, 0.1, 1, 10], # Regularization parameter
            'model__solver': ['liblinear', 'lbfgs'] # Solver algorithm
        }
    },
    'RandomForest': {
        'model': RandomForestClassifier(random_state=42, class_weight='balanced'),
        'params': {
            'model__n_estimators': [50, 100, 200, 500], # Number of trees in the j
            'model__max_depth': [None, 10, 20], # Maximum depth of the tree
        }
    },
    'SVR': {
        'model': SVR(kernel='rbf'), # Support Vector Regression with RBF kernel
```

```

        'params': {
            'model__C': [0.1, 1, 10], # Regularization parameter
            'model__gamma': ['scale', 'auto'], # Kernel coefficient
        },
    },
}

# Store the evaluation results
evaluation_results = {
    'Model': [],
    'Accuracy': [],
    'Precision': [],
    'Recall': [],
    'F1-Score': []
}

results = {}

for model_name, config in models.items(): # Loop through the models
    pipeline = Pipeline([ # Define a pipeline
        ('scaler', StandardScaler()), # Standardize features
        ('model', config['model']) # Add the model to the pipeline
    ])

    # GridSearchCV to find the best parameters
    grid = GridSearchCV(pipeline, config['params'], cv=5, scoring='accuracy', n_jobs=-1)
    grid.fit(X_train, y_train) # Fit the pipeline on training data

    # Store the results
    best_model = grid.best_estimator_ # Get the best model from grid search
    y_pred = best_model.predict(X_test) # Predict on the test set

    # For SVR, round predictions to calculate classification metrics
    if model_name == 'SVR':
        y_pred = y_pred.round()

    results[model_name] = grid.best_params_ # Save the best parameters

    # Calculate evaluation metrics
    accuracy = accuracy_score(y_test, y_pred)
    precision = precision_score(y_test, y_pred, average='binary')
    recall = recall_score(y_test, y_pred, average='binary')
    f1 = f1_score(y_test, y_pred, average='binary')

    # Save metrics in evaluation_results
    evaluation_results['Model'].append(model_name)
    evaluation_results['Accuracy'].append(accuracy)
    evaluation_results['Precision'].append(precision)
    evaluation_results['Recall'].append(recall)
    evaluation_results['F1-Score'].append(f1)

# Convert evaluation results to a DataFrame
evaluation_df = pd.DataFrame(evaluation_results)

# Display the evaluation metrics
print(evaluation_df)

```

	Model	Accuracy	Precision	Recall	F1-Score
0	LogisticRegression	0.765101	0.617021	0.630435	0.623656
1	RandomForest	0.785235	0.634615	0.717391	0.673469
2	SVR	0.798658	0.710526	0.586957	0.642857

```
c:\Daten\ADLS\ML\venv\Lib\site-packages\sklearn\model_selection\_search.py:1103: UserWarning: One or more of the test scores are non-finite: [nan nan nan nan nan nan]
warnings.warn(
```

- Der Code trainiert drei Modelle (Logistische Regression, Random Forest, und SVR) mit einer Pipeline, die die Features standardisiert, und optimiert ihre Hyperparameter mithilfe von GridSearchCV. Anschliessend werden die besten Modelle auf Testdaten angewendet, um Klassifikationsmetriken wie Genauigkeit, Präzision, Recall und F1-Score zu berechnen. Die Ergebnisse aller Modelle werden in einer übersichtlichen DataFrame-Tabelle gespeichert und angezeigt.

```
In [63]: # Visualize the ROC curves
plt.figure(figsize=(10, 8)) # Set figure size

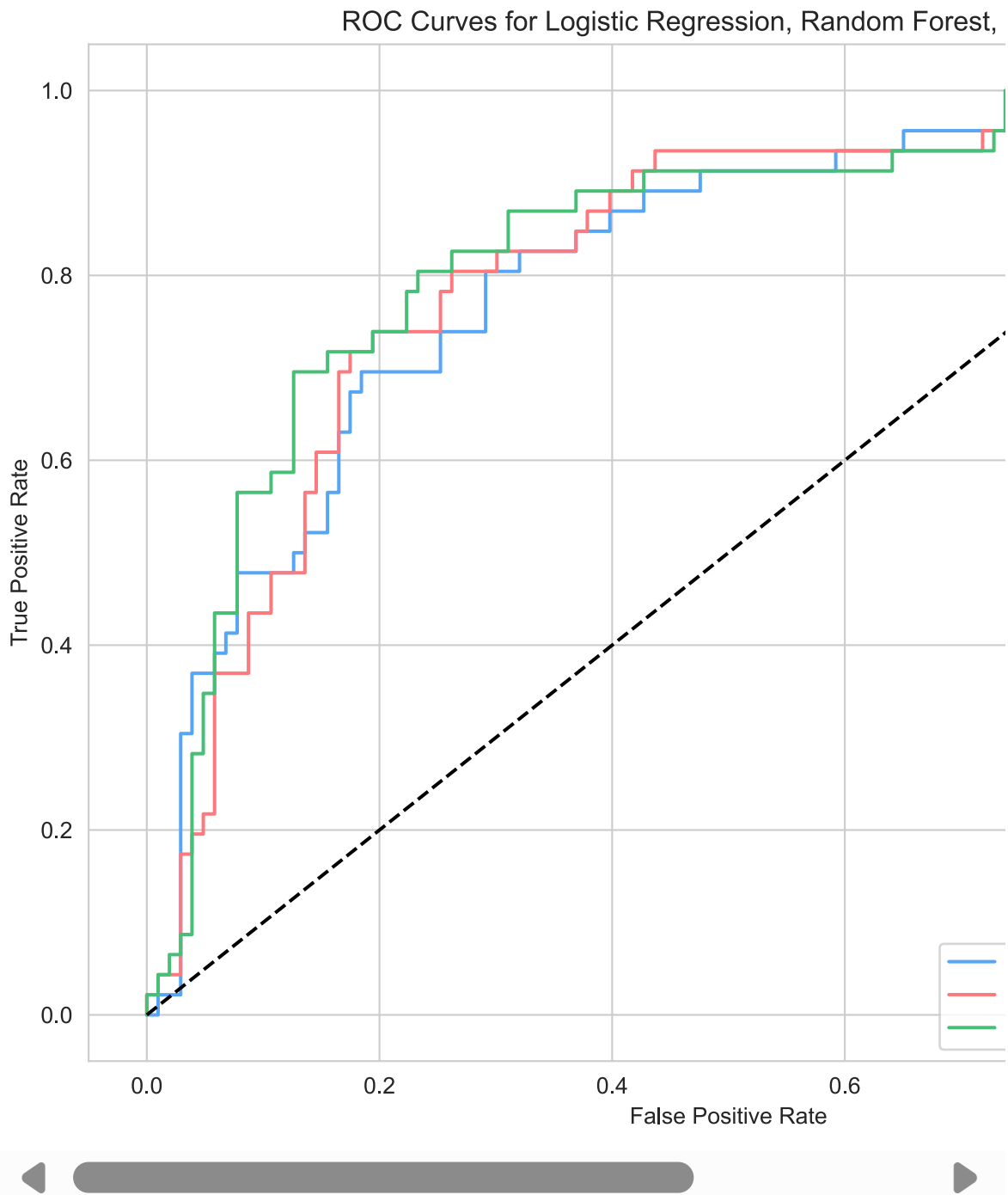
for model_name, config in models.items(): # Loop through the selected models
    pipeline = Pipeline([ # Define a pipeline
        ('scaler', StandardScaler()), # Standardize features
        ('model', config['model']) # Add the model
    ])
    pipeline.set_params(**results[model_name]) # Apply the best parameters
    pipeline.fit(X_train, y_train) # Fit the pipeline on the training data

    if model_name == 'SVR':
        # For SVR, normalize predictions for ROC computation
        y_pred_prob = pipeline.predict(X_test) # SVR outputs continuous values
        y_pred_prob = (y_pred_prob - y_pred_prob.min()) / (y_pred_prob.max() - y_pred_prob.min())
    else:
        # For Logistic Regression and Random Forest
        y_pred_prob = pipeline.predict_proba(X_test)[:, 1] # Predict probabilities

    fpr, tpr, _ = roc_curve(y_test, y_pred_prob) # Compute false positive and true positive rates
    roc_auc = auc(fpr, tpr) # Compute the area under the ROC curve

    plt.plot(fpr, tpr, label=f'{model_name} (AUC = {roc_auc:.2f})') # Plot the ROC curve

plt.plot([0, 1], [0, 1], 'k--') # Plot the random chance diagonal line
plt.xlabel('False Positive Rate') # Label for x-axis
plt.ylabel('True Positive Rate') # Label for y-axis
plt.title('ROC Curves for Logistic Regression, Random Forest, and SVR') # Title of the plot
plt.legend(loc='lower right') # Add Legend to the lower-right corner
plt.show() # Display the plot
```



- Der Code erstellt und trainiert Modelle (Logistische Regression, Random Forest, SVR) in Pipelines und berechnet die ROC-Kurven, um die Klassifikationsleistung zu vergleichen. Die Kurven und ihre AUC-Werte werden visualisiert, um die Vorhersagequalität der Modelle darzustellen.

Zusammenfassung der Erkenntnisse zur ROC-Kurve

- **Bestes Modell:** SVR erreicht die höchste AUC (0,84), was die beste Gesamtleistung bei der Unterscheidung zwischen den Klassen zeigt.
- **Vergleichbare Leistung:** Logistische Regression und Random Forest haben beide eine AUC von 0,81 und zeigen damit eine ähnliche, aber etwas geringere Klassifikationseffektivität.

```
In [64]: # Compute the confusion matrix
cm = confusion_matrix(y_test, y_pred)

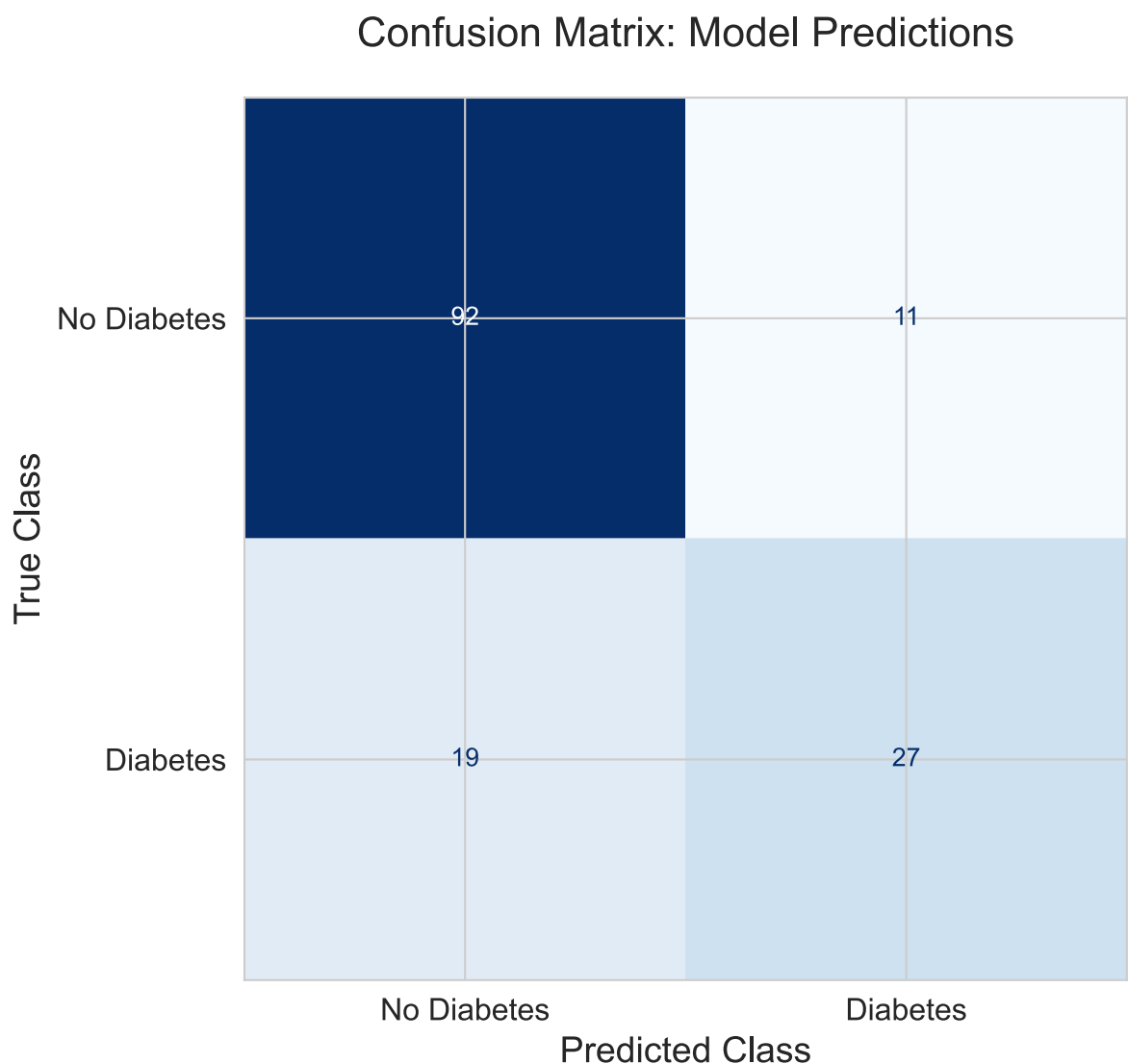
# Create the confusion matrix display
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=["No Diabetes",

# Plot the confusion matrix
fig, ax = plt.subplots(figsize=(8, 6)) # Larger size for better readability
disp.plot(cmap='Blues', ax=ax, values_format='d') # Set color scheme and format

# Add title and axis labels
ax.set_title("Confusion Matrix: Model Predictions", fontsize=16, pad=20)
ax.set_xlabel("Predicted Class", fontsize=14)
ax.set_ylabel("True Class", fontsize=14)
ax.tick_params(axis='both', which='major', labelsize=12)

# Customize tick labels
ax.set_xticklabels(["No Diabetes", "Diabetes"], fontsize=12)
ax.set_yticklabels(["No Diabetes", "Diabetes"], fontsize=12)

# Adjust layout and display the plot
plt.tight_layout()
plt.show()
```



Wichtige Erkenntnisse:

1. **True Negatives (TN):** 92 Fälle wurden korrekt als **Kein Diabetes** vorhergesagt.
 2. **False Positives (FP):** 11 Fälle wurden fälschlicherweise als **Diabetes** vorhergesagt, waren jedoch tatsächlich **Kein Diabetes**.
 3. **False Negatives (FN):** 19 Fälle von **Diabetes** wurden übersehen (als **Kein Diabetes** vorhergesagt).
 4. **True Positives (TP):** 27 Fälle wurden korrekt als **Diabetes** erkannt.
-

Diskussion und Fazit

Anleitung: Diskutiert hier kurz eure Erkenntnisse aus eurer Projektarbeit. Folgende Punkte müsst ihr adressieren:

- *Wie interpretiert eure Ergebnisse:*
 - *Welches Datenmodell funktioniert am besten?*
 - *Wie gut löst es das formulierte Problem?*
 - *Entsprechen die Ergebnisse euren Erwartungen?*
 - *Habt ihr Verbesserungsvorschläge für eure Datenmodelle?*
 - *Beschreibt eure Lernerlebnisse. Was waren eure wichtigsten Erkenntnisse im Verlauf dieses Projekts?*
1.
 - **Das Datenmodell welches am besten funktioniert ist Random Forest.**
 2.
 - **Random Forest löst das Problem ausgewogen. Verbesserungspotenzial ist noch da aber von den drei bzw anderen die ich ausprobiert habe und noch schlechter waren und nicht mehr drin sind hat es Random Forest okey gelöst, je nach dem wie streng man es sieht.**
 3.
 - **Also ich habe ein bisschen höhere Werte erwartet aber nach 2 Wochen ausprobieren und testen habe ich es dann auf den Ergebnissen belassen, da das die besten waren. Somit bin ich gespannt was ich in Zukunft noch lerne, um es zu optimieren.**
 4.
 - **Verbesserungsvorschläge habe ich keine für meine Datenmodelle sonst hätte ich sie ja schon angewendet. Wie gesagt habe ich sehr vieles versucht, bei dem ich bessere Werte erwartet habe aber nix dabei rausgekommen ist.**
 - 5.

- **Meine wichtigsten Erkenntnisse waren, dass es sehr wichtig ist einen guten Datensatz zu haben, da meiner viele Duplikate und 0 Werte hatte. Aber in Real Life bzw Life Science gibt es sicher oft 0 Werte aber ich denke nicht so viele Duplikate somit habe ich schon was dabei gelernt. Zudem war es gut vieles auszuprobieren und zu rechechieren so war der Lerneffekt gross von den theoretischen Konzepten diese in echt anzuwenden. Ebenfalls braucht es viel Zeit, wenn man es genau nehmen will und Ausdauer. Anzumerken ist noch, dass ich viel gelernt habe durch die Markdowns, welche ich als letztes hinzugefügt habe, um nochmals alles zu dokumentieren und für mich selbst zu festigen.**