Long_AI_Fundamentals

December 17, 2024

Open in Google Colab

0.1 Project #1 - AI Technical Principles & Basics

Johnathan Long
Predictive Model _____

Introduction

Artificial Intelligence has various use cases, such as prediction, detection, and generation. Its primary role is to identify correlations between variables. With a deeper understanding of these correlations, inferences can be made.

In this project, I want to compare two machine learning algorithms intended to perform predictive analysis. The focus is on understanding AI's internal mechanics and building a program to print metrics for further developments. Here I detail basic machine learning methods, intended to devliver an understanding of How AI works *"from the inside"* digestable for clientele, students, or researchers. Please note, that although basic methodology and approach may follow a general pattern, each machine learning model requires unique structure and architecture based upon the problem at hand.

We begin by introducing a "Heat Disease Prediction" dataset from Kaggle. The patient health data consist of features that can be used to predict if patients have high risk for heart disease.

[1]:
```python
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier, plot_tree
from sklearn.metrics import accuracy_score
import matplotlib.pyplot as plt
from sklearn.preprocessing import OneHotEncoder
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.model_selection import RandomizedSearchCV
from sklearn.model_selection import GridSearchCV
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, classification_report,␣
↪confusion_matrix
```

[2]:
```
file_path = '/content/Heart_Disease_Prediction.csv'
df = pd.read_csv(file_path)

print("First few rows of the dataset:")
df.head()
```

```
First few rows of the dataset:
```

[2]:
```
   Age  Sex  Chest pain type   BP  Cholesterol  FBS over 120  EKG results  \
0   70    1                4  130          322             0            2
1   67    0                3  115          564             0            2
2   57    1                2  124          261             0            0
3   64    1                4  128          263             0            0
4   74    0                2  120          269             0            2

   Max HR  Exercise angina  ST depression  Slope of ST  \
0     109                0            2.4            2
1     160                0            1.6            2
2     141                0            0.3            1
3     105                1            0.2            2
4     121                1            0.2            1

   Number of vessels fluro  Thallium Heart Disease
0                        3         3      Presence
1                        0         7       Absence
2                        0         7      Presence
3                        1         7       Absence
4                        1         3       Absence
```

[3]:
```
df.info()
```

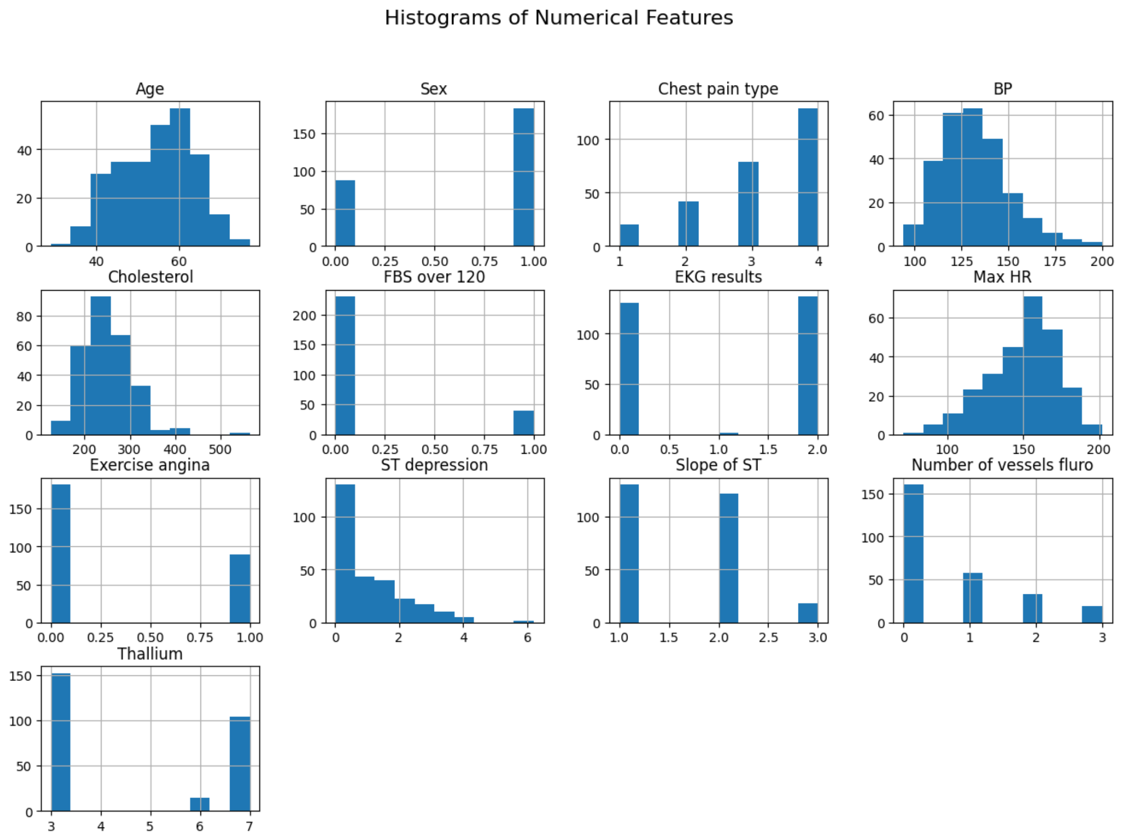
```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 270 entries, 0 to 269
Data columns (total 14 columns):
 #   Column                   Non-Null Count  Dtype
---  ------                   --------------  -----
 0   Age                      270 non-null    int64
 1   Sex                      270 non-null    int64
 2   Chest pain type          270 non-null    int64
 3   BP                       270 non-null    int64
 4   Cholesterol              270 non-null    int64
 5   FBS over 120             270 non-null    int64
 6   EKG results              270 non-null    int64
 7   Max HR                   270 non-null    int64
 8   Exercise angina          270 non-null    int64
 9   ST depression            270 non-null    float64
 10  Slope of ST              270 non-null    int64
 11  Number of vessels fluro  270 non-null    int64
 12  Thallium                 270 non-null    int64
 13  Heart Disease            270 non-null    object
dtypes: float64(1), int64(12), object(1)
memory usage: 29.7+ KB
```

Pre-processing

Preprocessing is a common step within most all data specific approaches ensuring the data is clean, structured and suitable for use. We visualize the first few rows of the dataset, datatype and features. There are 270 entries, 14 features, one of which is *"object"* type, or *"categorical"*. There are no missing values, indicating that we have a clean dataset.

Here the Heart Disease column is the *"target variable"*, or the variable that we are predicting. This column is seperated into *"Absence"* of heart disease (Negative Class) and *"Presence"* of heart disease(Positive Class). Because there exist only two possibilities, we can label this a *Binary Classification* problem. (predictions will either be "yes" or "no")

[4]:
```
# this is a histograms for all numerical columns
df.hist(figsize=(15, 10))
plt.suptitle('Histograms of Numerical Features', fontsize=16)
plt.show()
```



Plotting histograms is common practice and allows us to grasp insights about the data we will be analyzing. Inferences can be made such as - Most patients have fasting blood sugar levels not over 120 mg/dL.

Let's look at the "Sex" feature, where we may see an uneven distribution.

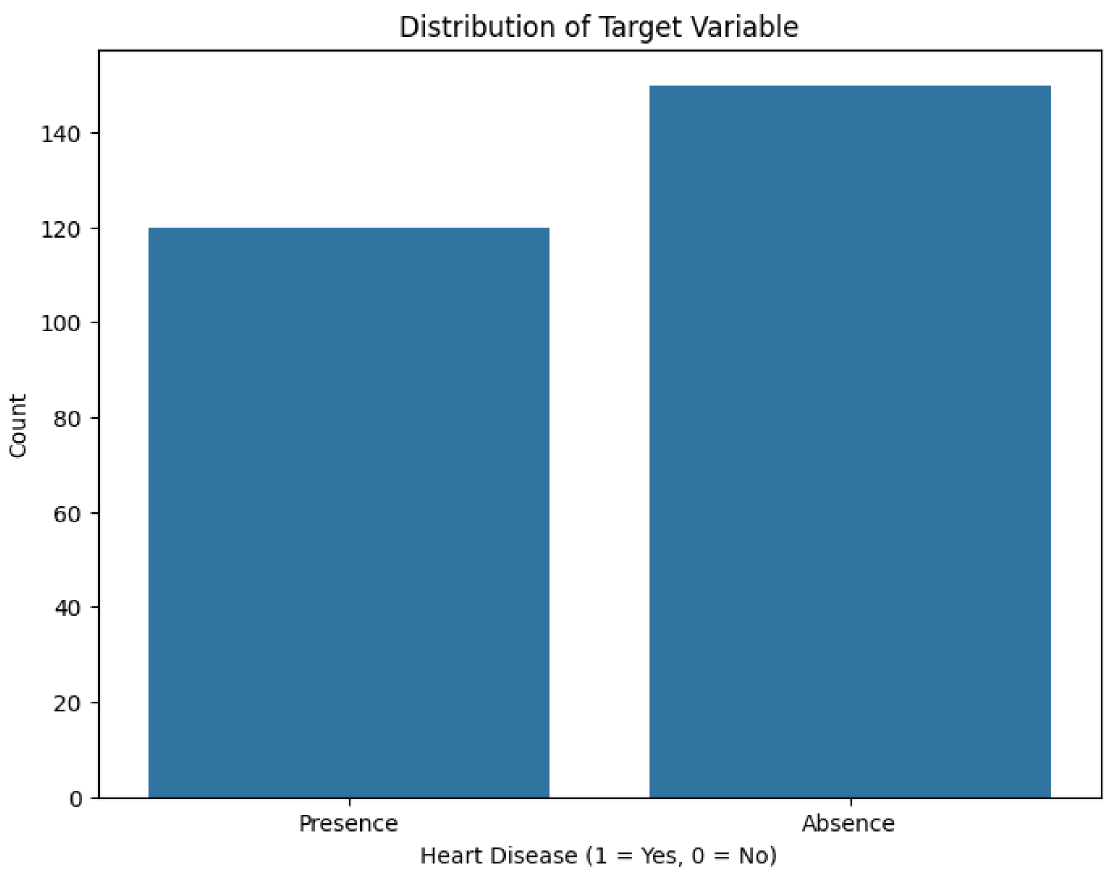
[5]:
```python
# I need to check the distribution of men versus women
gender_distribution = df['Sex'].value_counts()
print("Gender Distribution (0 = Female, 1 = Male):")
print(gender_distribution)
```

```
Gender Distribution (0 = Female, 1 = Male):
Sex
1    183
0     87
Name: count, dtype: int64
```

We see here that the number of Males outweigh the number of Females by almost 3 times as much. This is specific to the dataset. Whenever there are uneven distributions within the data, we can apply over/under sampling techniques to reduce the risk of bias. However this should be performed under careful consideration as it could result in misleading outputs.

Next we will see the number of heart disease vs no heart disease.

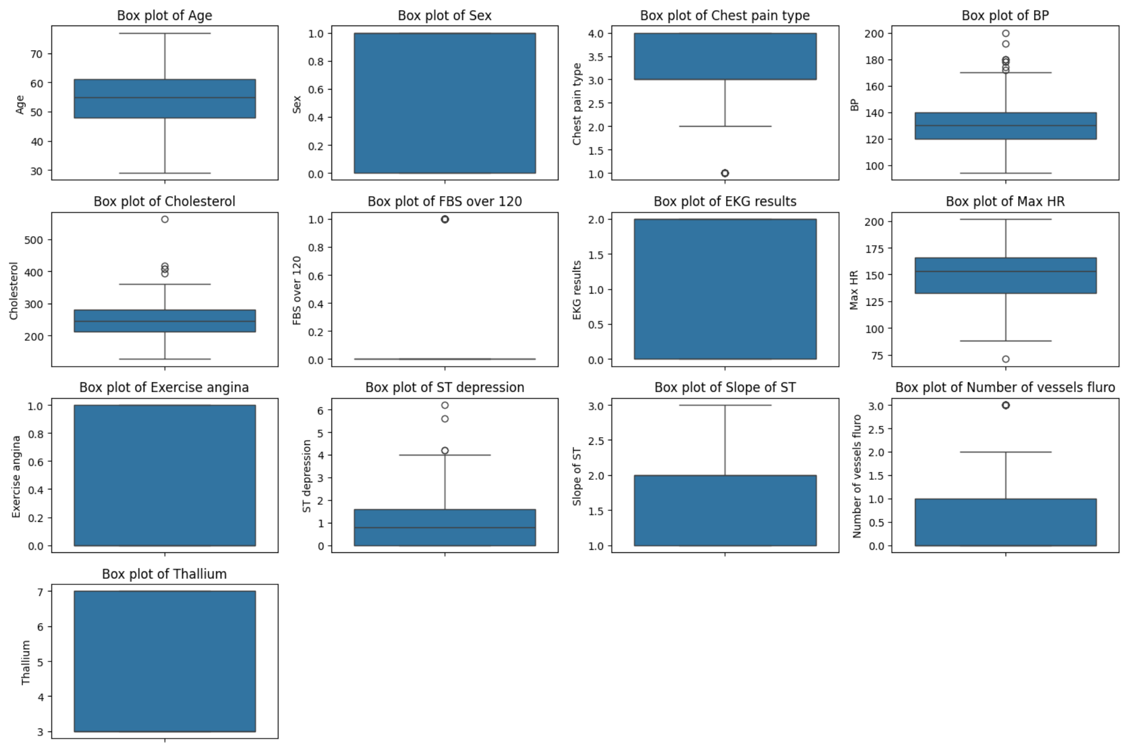
[6]:
```python
plt.figure(figsize=(8, 6))
sns.countplot(x='Heart Disease', data=df)
plt.title('Distribution of Target Variable')
plt.xlabel('Heart Disease (1 = Yes, 0 = No)')
plt.ylabel('Count')
plt.show()
```



Box plots are great for visualizing any outliers if present. Outliers can arrise when there are mismeasurement or innacuracies within data collection, however if the nature of the outliers lie within the true instance of the data it may be prefered to retain them. This does not hold true for binary (yes/no) features.

The specific outliers in Cholesterol, BP, and ST Depression will be capped at the 95th percentile to mitigate their impact and improve stability in the model.

[7]:
```
# We are going to perform correlation analysis to identify potential outliers␣
    ↪in the data
plt.figure(figsize=(15, 10))
for i, column in enumerate(df.columns[:-1], 1):
    plt.subplot(4, 4, i)
    sns.boxplot(y=df[column])
    plt.title(f'Box plot of {column}')
plt.tight_layout()
plt.show()
```



[8]:
```python
# This is the function to cap outliers at the 95th percentile
def cap_outliers(df, columns):
    for col in columns:
        upper_limit = df[col].quantile(0.95)
        df[col] = df[col].apply(lambda x: upper_limit if x > upper_limit else x)
    return df

# List of the specific columns to cap outliers
columns_to_cap = ['Cholesterol', 'BP', 'ST depression']

# Capping outliers
df = cap_outliers(df, columns_to_cap)

# Display summary statistics after capping
print("Summary statistics after capping outliers:")
df.describe()
```

```
Summary statistics after capping outliers:
```

[8]:
```
              Age         Sex  Chest pain type          BP  Cholesterol  \
count  270.000000  270.000000       270.000000  270.000000   270.000000
mean    54.433333    0.677778         3.174074  130.533333   247.106296
std      9.109067    0.468195         0.950090   15.905763    44.195526
```

```
min      29.000000    0.000000          1.000000    94.000000    126.000000
25%      48.000000    0.000000          3.000000   120.000000    213.000000
50%      55.000000    1.000000          3.000000   130.000000    245.000000
75%      61.000000    1.000000          4.000000   140.000000    280.000000
max      77.000000    1.000000          4.000000   160.000000    326.550000

       FBS over 120  EKG results      Max HR  Exercise angina  ST depression  \
count    270.000000   270.000000  270.000000       270.000000     270.000000
mean       0.148148     1.022222  149.677778         0.329630       1.011630
std        0.355906     0.997891   23.165717         0.470952       1.036166
min        0.000000     0.000000   71.000000         0.000000       0.000000
25%        0.000000     0.000000  133.000000         0.000000       0.000000
50%        0.000000     2.000000  153.500000         0.000000       0.800000
75%        0.000000     2.000000  166.000000         1.000000       1.600000
max        1.000000     2.000000  202.000000         1.000000       3.310000

       Slope of ST  Number of vessels fluro    Thallium
count   270.000000               270.000000  270.000000
mean      1.585185                 0.670370    4.696296
std       0.614390                 0.943896    1.940659
min       1.000000                 0.000000    3.000000
25%       1.000000                 0.000000    3.000000
50%       2.000000                 0.000000    3.000000
75%       2.000000                 1.000000    7.000000
max       3.000000                 3.000000    7.000000
```

Now to accurately predict the target variable (Heart Disease), we must convert it from a categorical variable to a numeric varibable. We do so here as Heart disease = 1 and No heart disease = 0.

[9]:
```python
# Create a copy of the original column for comparison
original_values = df['Heart Disease'].copy()

# Apply the mapping
df['Heart Disease'] = df['Heart Disease'].map({'Presence': 1, 'Absence': 0})
```

[10]:
```python
# Display changes
print("Changes in 'Heart Disease' column:")
pd.DataFrame({'Before': original_values, 'After': df['Heart Disease']})
```

```
Changes in 'Heart Disease' column:
```

[10]:
```
        Before  After
0     Presence      1
1      Absence      0
2     Presence      1
3      Absence      0
4      Absence      0
..         …      …
```

```
265   Absence      0
266   Absence      0
267   Absence      0
268   Absence      0
269  Presence      1

[270 rows x 2 columns]
```

CORRELATION MATRIX

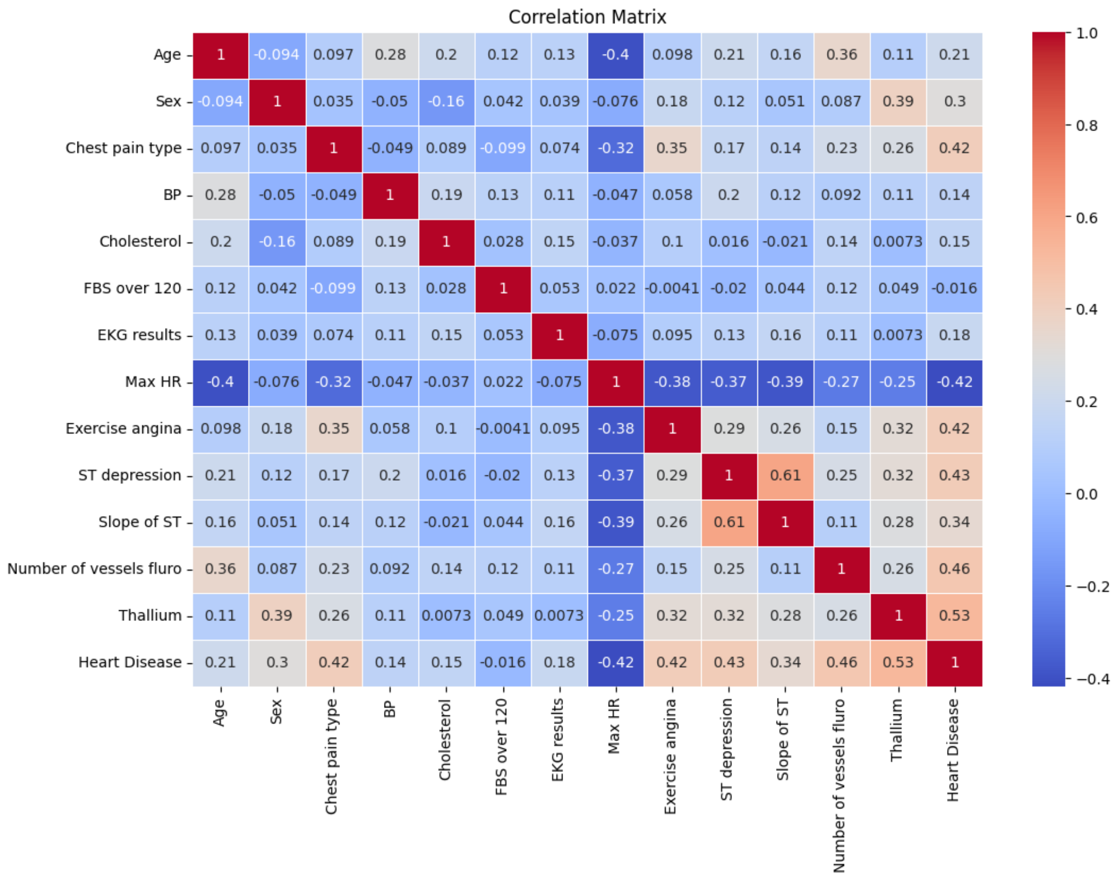
The correlation matrix presents insights into each features correlation with the target variable. Stronger correlations being darker while the polarity of the correlation depicting the direction of their relationship.

Examples:

Strong Positive Correlation:
Chest pain type and Heart Disease (0.42): *Certain types of chest pain are strongly associated with heart disease.*

Negative Correlation:
Max HR and Heart Disease (-0.42): *Lower maximum heart rates are associated with a higher likelihood of heart disease.*

[11]:
```
# And now calculate the correlation matrix
corr_matrix = df.corr()

plt.figure(figsize=(12, 8))
sns.heatmap(corr_matrix, annot=True, cmap='coolwarm', linewidths=0.5)
plt.title('Correlation Matrix')
plt.show()
```



	Age	Sex	Chest pain type	BP	Cholesterol	FBS over 120	EKG results	Max HR	Exercise angina	ST depression	Slope of ST	Number of vessels fluro	Thallium	Heart Disease
Age	1	-0.094	0.097	0.28	0.2	0.12	0.13	-0.4	0.098	0.21	0.16	0.36	0.11	0.21
Sex	-0.094	1	0.035	-0.05	-0.16	0.042	0.039	-0.076	0.18	0.12	0.051	0.087	0.39	0.3
Chest pain type	0.097	0.035	1	-0.049	0.089	-0.099	0.074	-0.32	0.35	0.17	0.14	0.23	0.26	0.42
BP	0.28	-0.05	-0.049	1	0.19	0.13	0.11	-0.047	0.058	0.2	0.12	0.092	0.11	0.14
Cholesterol	0.2	-0.16	0.089	0.19	1	0.028	0.15	-0.037	0.1	0.016	-0.021	0.14	0.0073	0.15
FBS over 120	0.12	0.042	-0.099	0.13	0.028	1	0.053	0.022	-0.0041	-0.02	0.044	0.12	0.049	-0.016
EKG results	0.13	0.039	0.074	0.11	0.15	0.053	1	-0.075	0.095	0.13	0.16	0.11	0.0073	0.18
Max HR	-0.4	-0.076	-0.32	-0.047	-0.037	0.022	-0.075	1	-0.38	-0.37	-0.39	-0.27	-0.25	-0.42
Exercise angina	0.098	0.18	0.35	0.058	0.1	-0.0041	0.095	-0.38	1	0.29	0.26	0.15	0.32	0.42
ST depression	0.21	0.12	0.17	0.2	0.016	-0.02	0.13	-0.37	0.29	1	0.61	0.25	0.32	0.43
Slope of ST	0.16	0.051	0.14	0.12	-0.021	0.044	0.16	-0.39	0.26	0.61	1	0.11	0.28	0.34
Number of vessels fluro	0.36	0.087	0.23	0.092	0.14	0.12	0.11	-0.27	0.15	0.25	0.11	1	0.26	0.46
Thallium	0.11	0.39	0.26	0.11	0.0073	0.049	0.0073	-0.25	0.32	0.32	0.28	0.26	1	0.53
Heart Disease	0.21	0.3	0.42	0.14	0.15	-0.016	0.18	-0.42	0.42	0.43	0.34	0.46	0.53	1

FEATURE ENGINEERING

Feature engineering involves improving the models predictive performance by selecting, creating, modifying, or transforming raw data. In other words, we will apply mathematical reasoning to distribute the data in a fashion that benefits the model. First, we *normalize* the values. Only the features Age, Blood Pressure, Cholesterol, Maximum Heart Rate, and ST depression were normalized because these are numerical features that have continuous values and varying scales (See histograms for reference). Normalizing them ensures that all numerical inputs have comparable ranges.

[12]:
```python
from sklearn.preprocessing import StandardScaler

# List of numerical columns to normalize
numerical_cols = ['Age', 'BP', 'Cholesterol', 'Max HR', 'ST depression']

# Initialize the StandardScaler
scaler = StandardScaler()

# Normalize the numerical columns
df[numerical_cols] = scaler.fit_transform(df[numerical_cols])
```

```
# Display the first few rows to confirm normalization
print("First few rows after normalization:")
df.head()
```

```
First few rows after normalization:
```

[12]:
```
        Age  Sex  Chest pain type        BP  Cholesterol  FBS over 120  \
0  1.712094    1                4 -0.033593     1.697746             0
1  1.382140    0                3 -0.978399     1.800889             0
2  0.282294    1                2 -0.411515     0.314953             0
3  1.052186    1                4 -0.159567     0.360290             0
4  2.152032    0                2 -0.663464     0.496303             0

   EKG results    Max HR  Exercise angina  ST depression  Slope of ST  \
0            2 -1.759208                0       1.342400            2
1            2  0.446409                0       0.568889            2
2            0 -0.375291                0      -0.688067            1
3            0 -1.932198                1      -0.784756            2
4            2 -1.240239                1      -0.784756            1

   Number of vessels fluro  Thallium  Heart Disease
0                        3         3              1
1                        0         7              0
2                        0         7              1
3                        1         7              0
4                        1         3              0
```

TRAIN-TEST-VALIDATION SPLIT

This code performs a *train-validation-test* split in two stages. First, the dataset is split into training (60%) and a temporary set (40%). Then, the temporary set is split equally into validation (20%) and test (20%) sets. This approach ensures that each dataset is distinct, avoiding data leakage and providing independent subsets for training, tuning, and evaluating the model. The method is versatile and suitable for datasets of any size, supporting rigorous model evaluation.

[13]:
```
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from xgboost import XGBClassifier
from sklearn.metrics import classification_report, confusion_matrix,␣
 ↪accuracy_score

# Split features from target variable
X = df.drop('Heart Disease', axis=1)
y = df['Heart Disease']

# Training (60%), validation (20%), and test (20%) sets
```

```
X_train, X_temp, y_train, y_temp = train_test_split(X, y, test_size=0.4,␣
 ↪random_state=42)
X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp, test_size=0.5,␣
 ↪random_state=42)
```

Random Forest:
The Random Forest model is an ensemble learning method that uses bagging (bootstrap aggregation) to build multiple decision trees independently and in parallel. Each tree is trained on a different random subset of the data, and their predictions are aggregated, typically using majority voting for classification tasks. This approach reduces variance and makes Random Forest robust to overfitting. In a binary classification task, Random Forest performs well on small to moderately sized datasets, requires minimal preprocessing, and handles noise effectively. However, it can be computationally expensive during inference, as predictions are aggregated from all trees. Random Forest also provides feature importance scores, which make it moderately interpretable, but the exact decision-making process is not always transparent.

The `.fit(X_train, y_train)` command trains the model, and often the most computationally expensive load within the program. As we have not yet made any adjustments or finetuning to the model, we can refer to it as a **base model**.

```
[14]: #  Random Forest model
rf_model = RandomForestClassifier(random_state=42)
rf_model.fit(X_train, y_train)
```

```
[14]: RandomForestClassifier(random_state=42)
```

```
[15]: # Making predictions with the Random Forest model
y_val_pred_rf = rf_model.predict(X_val)
```

Here we visualize the evaluation metrics directly. In other projects, we've seen "validation loss" and "accuracy" curves to analyze how the model is learning (training history learning curves). Due to the nature of this data, we use data size learning curves, that may not be as prevelant. We can conceptualize this as "What the model knows"(data-size) vs "How the model learns"(training-history). Provided certain real-life scenarios, the accuracy of the model may be cruicial (such as health care or financial industries where penalties for innacuracies are weighted heavily). In other circumstances, "how the model learns" may be a priority, especially in real-time scenarios where the model may interact with users.

When evaluating these metrics, think about taking a multiple choice test, and how eliminating all wrong answers is not necessarily the same method as accurately selecting the correct answer. Although both methods may result in a solution, the methodology is different, which here, is relevent.

Let's briefly review these metrics:

Accuracy:
Tthe model correctly predicted the class labels for about 70% of the validation samples. This is a decent baseline, but accuracy alone may not fully reflect the model's effectiveness.

Confusion Matrix:

The confusion matrix shows that the model is more effective at predicting class 0 (no heart disease) than class 1 (heart disease). This is evident from the higher number of true negatives and relatively fewer false positives compared to false negatives.

Class 0 (No Heart Disease):
Precision: 0.68 Of all predictions made for class 0, 68% were correct. This reflects a moderate level of precision.

Recall: 0.86
The model correctly identified 86% of all actual class 0 samples, indicating strong sensitivity to the negative class.

F1-Score: 0.76
Balances precision and recall, suggesting overall good performance for class 0.

Class 1 (Heart Disease):
Precision: 0.76 Of all predictions made for class 1, 76% were correct, which is higher than the negative class.

Recall: 0.52
The model identified only 52% of the actual class 1 samples, indicating weaker sensitivity to the positive class(heart disease) and a higher rate of false negatives.

F1-Score: 0.62
Reflects moderate performance for class 1 but lower than for class 0.

[16]:
```python
# Evaluate the Random Forest model on the validation set
print("Random Forest Model Evaluation on Validation Set:")
print("Accuracy:", accuracy_score(y_val, y_val_pred_rf))
print("Confusion Matrix:\n", confusion_matrix(y_val, y_val_pred_rf))
print("Classification Report:\n", classification_report(y_val, y_val_pred_rf))
```

```
Random Forest Model Evaluation on Validation Set:
Accuracy: 0.7037037037037037
Confusion Matrix:
 [[25  4]
 [12 13]]
Classification Report:
               precision    recall  f1-score   support

           0       0.68      0.86      0.76        29
           1       0.76      0.52      0.62        25

    accuracy                           0.70        54
   macro avg       0.72      0.69      0.69        54
weighted avg       0.72      0.70      0.69        54
```

We now apply hyperparameter tuning, specifying the constraits of the base model to explore if we can achieve improvments. Here instead of modifying hyperparameters mannually, we automate this process by applying a *Grid Search* which will find the best parameters for us by trainnig each possible combination within the alloted ranges. The `.fit` function is called here to train through multiple cycles, expect an increase in load.

[17]:
```
# Defining parameter grid for Random Forest with more options
param_grid_rf = {
    'n_estimators': [100, 200, 300 ],
    'max_depth': [None, 10, 20, 30 ],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4],
    'bootstrap': [True, False]
}

# Initialize the Random Forest model
rf_model = RandomForestClassifier(random_state=42)

# Perform Grid Search
grid_search = GridSearchCV(estimator=rf_model, param_grid=param_grid_rf,
                           cv=5, n_jobs=-1, scoring='accuracy', verbose=1)

# Fit GridSearchCV
grid_search.fit(X_train, y_train)
```

```
Fitting 5 folds for each of 216 candidates, totalling 1080 fits
```

[17]:
```
GridSearchCV(cv=5, estimator=RandomForestClassifier(random_state=42), n_jobs=-1,
             param_grid={'bootstrap': [True, False],
                         'max_depth': [None, 10, 20, 30],
                         'min_samples_leaf': [1, 2, 4],
                         'min_samples_split': [2, 5, 10],
                         'n_estimators': [100, 200, 300]},
             scoring='accuracy', verbose=1)
```

[18]:
```
# Best parameters and estimator
best_params_rf = grid_search.best_params_
best_rf_model = grid_search.best_estimator_

print("Best Parameters Found by Grid Search:")
print(best_params_rf)
```

```
Best Parameters Found by Grid Search:
{'bootstrap': True, 'max_depth': None, 'min_samples_leaf': 2,
'min_samples_split': 10, 'n_estimators': 100}
```

After locating the best parameters we use those same parameters to train a model.

[19]:
```
# Train the model with the best params
best_model_rf = RandomForestClassifier(**best_params_rf, random_state=42)
best_model_rf.fit(X_train, y_train)
```

```
# Predict on validation set using the best model
rf_val_preds = best_rf_model.predict(X_val)
```

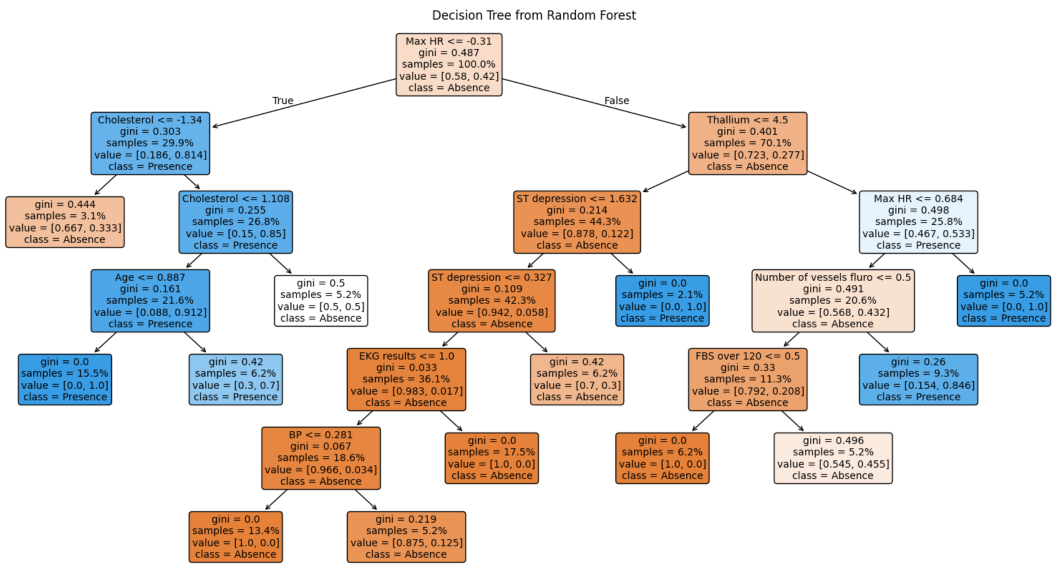
[20]:
```
print("Validation Results with Optimized Random Forest:")
print("Accuracy:", accuracy_score(y_val, rf_val_preds))
print("Confusion Matrix:\n", confusion_matrix(y_val, rf_val_preds))
print("Classification Report:\n", classification_report(y_val, rf_val_preds))
```

```
Validation Results with Optimized Random Forest:
Accuracy: 0.6851851851851852
Confusion Matrix:
 [[24  5]
 [12 13]]
Classification Report:
               precision    recall  f1-score   support

           0       0.67      0.83      0.74        29
           1       0.72      0.52      0.60        25

    accuracy                           0.69        54
   macro avg       0.69      0.67      0.67        54
weighted avg       0.69      0.69      0.68        54
```

The decision tree from the random forest gives us insight into how it classifies its decisions. Visualizing the output provides more transparency.

[21]:
```
# Access a single tree from the trained Random Forest
single_tree = best_rf_model.estimators_[6]  # Select the first tree
```

[22]:
```
from sklearn.tree import plot_tree
import matplotlib.pyplot as plt

# Plot the selected tree
plt.figure(figsize=(20, 10))
plot_tree(
    single_tree,
    feature_names=X.columns,  # Column names for better interpretability
    class_names=['Absence', 'Presence'],  # Target class names
    filled=True,  # Color nodes based on class
    rounded=True,  # Rounded nodes for aesthetics
    proportion=True,  # Scale nodes proportionally to samples
    fontsize=10  # Adjust font size for readability
)
plt.title("Decision Tree from Random Forest")
plt.show()
```


Decision Tree from Random Forest
Max HR <= -0.31 gini = 0.487 samples = 100.0% value = [0.58, 0.42] class = Absence
True
False
Cholesterol <= -1.34 gini = 0.303 samples = 29.9% value = [0.186, 0.814] class = Presence
Thallium <= 4.5 gini = 0.401 samples = 70.1% value = [0.723, 0.277] class = Absence
gini = 0.444 samples = 3.1% value = [0.667, 0.333] class = Absence
Cholesterol <= 1.108 gini = 0.255 samples = 26.8% value = [0.15, 0.85] class = Presence
ST depression <= 1.632 gini = 0.214 samples = 44.3% value = [0.878, 0.122] class = Absence
Max HR <= 0.684 gini = 0.498 samples = 25.8% value = [0.467, 0.533] class = Presence
Age <= 0.887 gini = 0.161 samples = 21.6% value = [0.088, 0.912] class = Presence
gini = 0.5 samples = 5.2% value = [0.5, 0.5] class = Absence
ST depression <= 0.327 gini = 0.109 samples = 42.3% value = [0.942, 0.058] class = Absence
gini = 0.0 samples = 2.1% value = [0.0, 1.0] class = Presence
Number of vessels fluro <= 0.5 gini = 0.491 samples = 20.6% value = [0.568, 0.432] class = Absence
gini = 0.0 samples = 5.2% value = [0.0, 1.0] class = Presence
gini = 0.0 samples = 15.5% value = [0.0, 1.0] class = Presence
gini = 0.42 samples = 6.2% value = [0.3, 0.7] class = Presence
EKG results <= 1.0 gini = 0.033 samples = 36.1% value = [0.983, 0.017] class = Absence
gini = 0.42 samples = 6.2% value = [0.7, 0.3] class = Absence
FBS over 120 <= 0.5 gini = 0.33 samples = 11.3% value = [0.792, 0.208] class = Absence
gini = 0.26 samples = 9.3% value = [0.154, 0.846] class = Presence
BP <= 0.281 gini = 0.067 samples = 18.6% value = [0.966, 0.034] class = Absence
gini = 0.0 samples = 17.5% value = [1.0, 0.0] class = Absence
gini = 0.0 samples = 6.2% value = [1.0, 0.0] class = Absence
gini = 0.496 samples = 5.2% value = [0.545, 0.455] class = Absence
gini = 0.0 samples = 13.4% value = [1.0, 0.0] class = Absence
gini = 0.219 samples = 5.2% value = [0.875, 0.125] class = Absence

---

XGBoost Model

Extreme Gradient Boosting is a method where decision trees are built sequentially, with each tree correcting the errors of the previous ones by optimizing a loss function. This makes XGBoost particularly effective at capturing complex patterns in the data. In a binary classification task, XGBoost excels at delivering high accuracy on large and complex datasets, often outperforming other models with proper hyperparameter tuning. Compared to Random Forest, XGBoost is computationally more intensive to train due to its sequential learning, but it often requires fewer trees for inference, resulting in faster predictions. While XGBoost is less interpretable due to its advanced optimizations and sequential process, it is a powerful tool for scenarios requiring high precision such as within the healthcare sector.

[23]:
```
# Initialize the base XGBoost Classifier
xgb_model = XGBClassifier(random_state=42, use_label_encoder=False,␣
 ↪eval_metric='logloss')

# Train the model on the training set
xgb_model.fit(X_train, y_train)
```

```
/usr/local/lib/python3.10/dist-packages/xgboost/core.py:158: UserWarning:
[04:35:43] WARNING: /workspace/src/learner.cc:740:
Parameters: { "use_label_encoder" } are not used.

  warnings.warn(smsg, UserWarning)
```

[23]:
```
XGBClassifier(base_score=None, booster=None, callbacks=None,
              colsample_bylevel=None, colsample_bynode=None,
              colsample_bytree=None, device=None, early_stopping_rounds=None,
              enable_categorical=False, eval_metric='logloss',
              feature_types=None, gamma=None, grow_policy=None,
              importance_type=None, interaction_constraints=None,
              learning_rate=None, max_bin=None, max_cat_threshold=None,
              max_cat_to_onehot=None, max_delta_step=None, max_depth=None,
              max_leaves=None, min_child_weight=None, missing=nan,
              monotone_constraints=None, multi_strategy=None, n_estimators=None,
              n_jobs=None, num_parallel_tree=None, random_state=42, …)
```

XGBoost provides marginally better overall performance than Random Forest, with a higher accuracy, improved precision for the positive class, and better recall for the negative class. These results make XGBoost a better choice, particularly if minimizing false positives or achieving more balanced performance is a priority. However, both models share the limitation of weak recall for the positive class, suggesting the need for further adjustments,

[24]:
```
# Predict on the validation set
xgb_val_preds = xgb_model.predict(X_val)

# Evaluate the base model
print("Base XGBoost Validation Results:")
print("Accuracy:", accuracy_score(y_val, xgb_val_preds))
print("Confusion Matrix:\n", confusion_matrix(y_val, xgb_val_preds))
print("Classification Report:\n", classification_report(y_val, xgb_val_preds))
```

```
Base XGBoost Validation Results:
Accuracy: 0.7222222222222222
Confusion Matrix:
 [[26  3]
 [12 13]]
Classification Report:
               precision    recall  f1-score   support

           0       0.68      0.90      0.78        29
           1       0.81      0.52      0.63        25

    accuracy                           0.72        54
   macro avg       0.75      0.71      0.71        54
weighted avg       0.74      0.72      0.71        54
```

Here, we automate hyperparameter tuning once again. However, instead of performing a GridSearch, which trains on every possible combination of parameters, we use a RandomizedSearch. RandomizedSearch trains on a limited number of randomly selected parameter combinations, retaining the best outcome.

[25]:
```
from xgboost import XGBClassifier
from sklearn.model_selection import RandomizedSearchCV
```

```
import numpy as np

# Define the parameter distribution (same as before)
param_dist_xgb = {
    'n_estimators': [100, 200, 300, 400, 500],
    'max_depth': [3, 4, 5, 6, 7, 8, 9, 10],
    'learning_rate': [0.01, 0.05, 0.1, 0.2, 0.3],
    'subsample': [0.6, 0.7, 0.8, 0.9, 1.0],
    'colsample_bytree': [0.6, 0.7, 0.8, 0.9, 1.0],
    'gamma': [0, 0.1, 0.2, 0.3, 0.4, 0.5]
}

# Initialize the XGBoost model
xgb_model = XGBClassifier(
    random_state=42,
    use_label_encoder=False,
    eval_metric='logloss',
)

# Perform RandomizedSearchCV with a fixed number of iterations (e.g., 50)
random_search = RandomizedSearchCV(
    estimator=xgb_model,
    param_distributions=param_dist_xgb,  # Randomized sampling from the␣
  ↪specified distributions
    n_iter=50,  # Number of iterations for random search
    cv=5,
    scoring='accuracy',
    n_jobs=-1,
    verbose=1,
    random_state=42
)
```

[26]:
```
# Fit RandomizedSearchCV
random_search.fit(X_train, y_train)
```

```
Fitting 5 folds for each of 50 candidates, totalling 250 fits
```

```
/usr/local/lib/python3.10/dist-packages/xgboost/core.py:158: UserWarning:
[04:36:04] WARNING: /workspace/src/learner.cc:740:
Parameters: { "use_label_encoder" } are not used.

  warnings.warn(smsg, UserWarning)
```

[26]:
```
RandomizedSearchCV(cv=5,
                   estimator=XGBClassifier(base_score=None, booster=None,
                                           callbacks=None,
                                           colsample_bylevel=None,
                                           colsample_bynode=None,
                                           colsample_bytree=None, device=None,
                                           early_stopping_rounds=None,
                                           enable_categorical=False,
                                           eval_metric='logloss',
                                           feature_types=None, gamma=None,
                                           grow_policy=None,
                                           importance_type=None,
                                           interaction_constraints=None,
                                           learning…
                                           n_estimators=None, n_jobs=None,
                                           num_parallel_tree=None,
                                           random_state=42, …),
                   n_iter=50, n_jobs=-1,
                   param_distributions={'colsample_bytree': [0.6, 0.7, 0.8, 0.9,
                                                             1.0],
                                        'gamma': [0, 0.1, 0.2, 0.3, 0.4, 0.5],
                                        'learning_rate': [0.01, 0.05, 0.1, 0.2,
                                                          0.3],
                                        'max_depth': [3, 4, 5, 6, 7, 8, 9, 10],
                                        'n_estimators': [100, 200, 300, 400,
                                                         500],
                                        'subsample': [0.6, 0.7, 0.8, 0.9, 1.0]},
                   random_state=42, scoring='accuracy', verbose=1)
```

[27]:
```python
# Best parameters and estimator from random search
best_params = random_search.best_params_
best_xgb_model = random_search.best_estimator_

# Print the best parameters found by RandomizedSearchCV
print("Best Parameters Found by RandomizedSearchCV:")
print(best_params)
```

```
Best Parameters Found by RandomizedSearchCV:
{'subsample': 0.6, 'n_estimators': 200, 'max_depth': 3, 'learning_rate': 0.01,
'gamma': 0.3, 'colsample_bytree': 0.6}
```

Using the best parameters recognized by the RandomizedSearch, we train an XGB model. The model shows improvements in accuracy, precision, and recall for class 0 but continues to struggle with recall for class 1. This model might be acceptable if minimizing false positives is critical. However, for this specific case scenario we are prioritizing correct identification of the positive class (We want to correctly diagnose as many people with heart disease as possible) therefore further refinement is necessary.

[28]:
```python
# Train the XGBoost model with the best parameters
best_xgb_model.fit(X_train, y_train)

# Make predictions on the validation set
y_pred_xgb = best_xgb_model.predict(X_val)
```

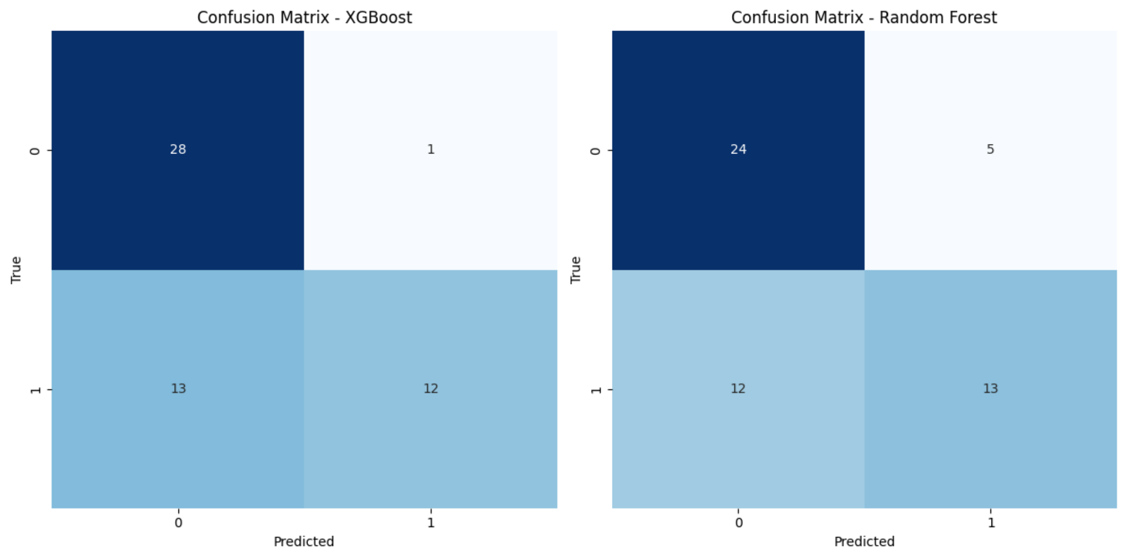
```
# Evaluate the predictions
accuracy = accuracy_score(y_val, y_pred_xgb)
print(f"Accuracy on Validation Set: {accuracy:.4f}")

# Print the classification report
print("\nClassification Report:")
print(classification_report(y_val, y_pred_xgb))

# Print the confusion matrix
print("\nConfusion Matrix:")
print(confusion_matrix(y_val, y_pred_xgb))
```

```
/usr/local/lib/python3.10/dist-packages/xgboost/core.py:158: UserWarning:
[04:36:04] WARNING: /workspace/src/learner.cc:740:
Parameters: { "use_label_encoder" } are not used.

  warnings.warn(smsg, UserWarning)
```

```
Accuracy on Validation Set: 0.7407

Classification Report:
              precision    recall  f1-score   support

           0       0.68      0.97      0.80        29
           1       0.92      0.48      0.63        25

    accuracy                           0.74        54
   macro avg       0.80      0.72      0.72        54
weighted avg       0.79      0.74      0.72        54


Confusion Matrix:
[[28  1]
 [13 12]]
```

[29]:
```
# Make predictions for both models

# Generate confusion matrices for both models
cm_xgb = confusion_matrix(y_val, y_pred_xgb)
cm_rf = confusion_matrix(y_val, rf_val_preds)

# Create a 1x2 subplot to display both confusion matrices side by side
fig, axes = plt.subplots(1, 2, figsize=(12, 6))

# Plot the XGBoost confusion matrix
sns.heatmap(cm_xgb, annot=True, fmt='d', cmap='Blues', ax=axes[0], cbar=False)
axes[0].set_title('Confusion Matrix - XGBoost')
```

```
axes[0].set_xlabel('Predicted')
axes[0].set_ylabel('True')

# Plot the Random Forest confusion matrix
sns.heatmap(cm_rf, annot=True, fmt='d', cmap='Blues', ax=axes[1], cbar=False)
axes[1].set_title('Confusion Matrix - Random Forest')
axes[1].set_xlabel('Predicted')
axes[1].set_ylabel('True')

# Display the plot
plt.tight_layout()
plt.show()
```



Reading the confusion matrix:
NO Heart Disease: 0
Heart Disease: 1

The XGBoost model performs slightly better than the RF model, but both model behave similarly in inaccurately predicting the positive class. We see that the majority of instances that were predicted *class 0*, were actually *class 0*, however we do see inaccuracies where the models predicted *class 0*, the true instance was *class 1*. An imrpoved confusion matrix would present a strong diagnal layout, meaning we would want the bottom left square to appear much lighter and the bottom right square to be much darker

```
[30]: from sklearn.metrics import precision_score, recall_score, f1_score,␣
        ↪roc_auc_score, log_loss

      # Calculate the metrics for both models
      accuracy_xgb = accuracy_score(y_val, y_pred_xgb)
```

```
accuracy_rf = accuracy_score(y_val, rf_val_preds)

precision_xgb = precision_score(y_val, y_pred_xgb)
precision_rf = precision_score(y_val, rf_val_preds)

recall_xgb = recall_score(y_val, y_pred_xgb)
recall_rf = recall_score(y_val, rf_val_preds)

f1_xgb = f1_score(y_val, y_pred_xgb)
f1_rf = f1_score(y_val, rf_val_preds)

auc_xgb = roc_auc_score(y_val, best_xgb_model.predict_proba(X_val)[:, 1])
auc_rf = roc_auc_score(y_val, best_rf_model.predict_proba(X_val)[:, 1])

log_loss_xgb = log_loss(y_val, best_xgb_model.predict_proba(X_val))
log_loss_rf = log_loss(y_val, best_rf_model.predict_proba(X_val))

# Print the results
print("XGBoost Model Metrics:")
print(f"Accuracy: {accuracy_xgb:.4f}")
print(f"Precision: {precision_xgb:.4f}")
print(f"Recall: {recall_xgb:.4f}")
print(f"F1-Score: {f1_xgb:.4f}")
print(f"AUC: {auc_xgb:.4f}")
print(f"Log Loss: {log_loss_xgb:.4f}")

print("\nRandom Forest Model Metrics:")
print(f"Accuracy: {accuracy_rf:.4f}")
print(f"Precision: {precision_rf:.4f}")
print(f"Recall: {recall_rf:.4f}")
print(f"F1-Score: {f1_rf:.4f}")
print(f"AUC: {auc_rf:.4f}")
print(f"Log Loss: {log_loss_rf:.4f}")
```

```
XGBoost Model Metrics:
Accuracy: 0.7407
Precision: 0.9231
Recall: 0.4800
F1-Score: 0.6316
AUC: 0.7600
Log Loss: 0.5714

Random Forest Model Metrics:
Accuracy: 0.6852
Precision: 0.7222
Recall: 0.5200
F1-Score: 0.6047
```

```
AUC: 0.7586
Log Loss: 0.5945
```

Conclusion
Both models show functionality as their predictions are moderately accurate, XGBoost moreso than the Random Forest Model. Hyperparameter tuning slightly improved model performance for both models although they both continued to struggle with the recall for the positive class. This could potentially result in a high false-negative rate, which is critical for the healthcare sector. This is likely due to class imbalance (More cases with heart disease than without). Further research would suggest that this can be address with over/undersampling techniques, including using **SMOTE**.

As detailed in the last cell below, we see instances where a more simple, less complex model can improve accuracy at the cost of recall and additional metrics. It is important to keep in mind that the relavance of the accuracy rate is subject to the nature of the task. The objective is to locate the best performing model specific to our task.

Future improvements would include adjusting weights, or decision thresholds in hopes to balance the trade-off between percision and recall. Also exploring feature importance to identify possible refinments for preprocessing.

[45]:
```python
rf2 = RandomForestClassifier(n_estimators=50, random_state=2)

rf2.fit(X_train, y_train)

y_pred2 = rf2.predict(X_test)

print(f"Accuracy:{accuracy_score(y_test, y_pred2)}")
print(f"Percision: {precision_score(y_val, y_pred2)}")
print(f"Recall: {recall_score(y_val, y_pred2)}")
print(f"F1 Score: {f1_score(y_val, y_pred2)}")
```

```
Accuracy:0.7777777777777778
Percision: 0.5625
Recall: 0.36
F1 Score: 0.43902439024390244
```

[53]:
```
#sudo install install pandoc
#pip install pypandoc
```

[52]:
```
#apt-get update
#apt-get install texlive-xetex texlive-fonts-recommended texlive-plain-generic
```

[55]:
```
# prompt: Let's convert this notebook to pdf using !jupyter notebook


!apt-get install texlive-xetex texlive-fonts-recommended texlive-plain-generic
```

```
Reading package lists… Done
Building dependency tree… Done
```

```
Reading state information… Done
texlive-fonts-recommended is already the newest version (2021.20220204-1).
texlive-plain-generic is already the newest version (2021.20220204-1).
texlive-xetex is already the newest version (2021.20220204-1).
0 upgraded, 0 newly installed, 0 to remove and 56 not upgraded.
```

[54]:
```
import pypandoc

!jupyter nbconvert --to pdf '/content/drive/MyDrive/Colab Notebooks/
 ↪Long_AI_Fundamentals.ipynb'
```

```
This application is used to convert notebook files (*.ipynb)
        to various other formats.

        WARNING: THE COMMANDLINE INTERFACE MAY CHANGE IN FUTURE RELEASES.

Options
=======
The options below are convenience aliases to configurable class-options,
as listed in the "Equivalent to" description-line of the aliases.
To see all configurable class-options for some <cmd>, use:
    <cmd> --help-all

--debug
    set log level to logging.DEBUG (maximize logging output)
    Equivalent to: [--Application.log_level=10]
--show-config
    Show the application's configuration (human-readable format)
    Equivalent to: [--Application.show_config=True]
--show-config-json
    Show the application's configuration (json format)
    Equivalent to: [--Application.show_config_json=True]
--generate-config
    generate default config file
    Equivalent to: [--JupyterApp.generate_config=True]
-y
    Answer yes to any questions instead of prompting.
    Equivalent to: [--JupyterApp.answer_yes=True]
--execute
    Execute the notebook prior to export.
    Equivalent to: [--ExecutePreprocessor.enabled=True]
--allow-errors
    Continue notebook execution even if one of the cells throws an error and
include the error message in the cell output (the default behaviour is to abort
conversion). This flag is only relevant if '--execute' was specified, too.
    Equivalent to: [--ExecutePreprocessor.allow_errors=True]
--stdin
    read a single notebook file from stdin. Write the resulting notebook with
```

```
default basename 'notebook.*'
    Equivalent to: [--NbConvertApp.from_stdin=True]
--stdout
    Write notebook output to stdout instead of files.
    Equivalent to: [--NbConvertApp.writer_class=StdoutWriter]
--inplace
    Run nbconvert in place, overwriting the existing notebook (only
            relevant when converting to notebook format)
    Equivalent to: [--NbConvertApp.use_output_suffix=False
--NbConvertApp.export_format=notebook --FilesWriter.build_directory=]
--clear-output
    Clear output of current file and save in place,
            overwriting the existing notebook.
    Equivalent to: [--NbConvertApp.use_output_suffix=False
--NbConvertApp.export_format=notebook --FilesWriter.build_directory=
--ClearOutputPreprocessor.enabled=True]
--coalesce-streams
    Coalesce consecutive stdout and stderr outputs into one stream (within each
cell).
    Equivalent to: [--NbConvertApp.use_output_suffix=False
--NbConvertApp.export_format=notebook --FilesWriter.build_directory=
--CoalesceStreamsPreprocessor.enabled=True]
--no-prompt
    Exclude input and output prompts from converted document.
    Equivalent to: [--TemplateExporter.exclude_input_prompt=True
--TemplateExporter.exclude_output_prompt=True]
--no-input
    Exclude input cells and output prompts from converted document.
            This mode is ideal for generating code-free reports.
    Equivalent to: [--TemplateExporter.exclude_output_prompt=True
--TemplateExporter.exclude_input=True
--TemplateExporter.exclude_input_prompt=True]
--allow-chromium-download
    Whether to allow downloading chromium if no suitable version is found on the
system.
    Equivalent to: [--WebPDFExporter.allow_chromium_download=True]
--disable-chromium-sandbox
    Disable chromium security sandbox when converting to PDF..
    Equivalent to: [--WebPDFExporter.disable_sandbox=True]
--show-input
    Shows code input. This flag is only useful for dejavu users.
    Equivalent to: [--TemplateExporter.exclude_input=False]
--embed-images
    Embed the images as base64 dataurls in the output. This flag is only useful
for the HTML/WebPDF/Slides exports.
    Equivalent to: [--HTMLExporter.embed_images=True]
--sanitize-html
    Whether the HTML in Markdown cells and cell outputs should be sanitized..
    Equivalent to: [--HTMLExporter.sanitize_html=True]
--log-level=<Enum>
    Set the log level by value or name.
    Choices: any of [0, 10, 20, 30, 40, 50, 'DEBUG', 'INFO', 'WARN', 'ERROR',
'CRITICAL']
    Default: 30
    Equivalent to: [--Application.log_level]
--config=<Unicode>
    Full path of a config file.
    Default: ''
    Equivalent to: [--JupyterApp.config_file]
--to=<Unicode>
    The export format to be used, either one of the built-in formats
            ['asciidoc', 'custom', 'html', 'latex', 'markdown', 'notebook',
'pdf', 'python', 'qtpdf', 'qtpng', 'rst', 'script', 'slides', 'webpdf']
            or a dotted object name that represents the import path for an
            ``Exporter`` class
    Default: ''
    Equivalent to: [--NbConvertApp.export_format]
--template=<Unicode>
    Name of the template to use
    Default: ''
    Equivalent to: [--TemplateExporter.template_name]
--template-file=<Unicode>
    Name of the template file to use
    Default: None
    Equivalent to: [--TemplateExporter.template_file]
--theme=<Unicode>
    Template specific theme(e.g. the name of a JupyterLab CSS theme distributed
    as prebuilt extension for the lab template)
    Default: 'light'
    Equivalent to: [--HTMLExporter.theme]
--sanitize_html=<Bool>
    Whether the HTML in Markdown cells and cell outputs should be sanitized.This
    should be set to True by nbviewer or similar tools.
    Default: False
    Equivalent to: [--HTMLExporter.sanitize_html]
--writer=<DottedObjectName>
    Writer class used to write the
                                    results of the conversion
    Default: 'FilesWriter'
    Equivalent to: [--NbConvertApp.writer_class]
--post=<DottedOrNone>
    PostProcessor class used to write the
                                    results of the conversion
    Default: ''
    Equivalent to: [--NbConvertApp.postprocessor_class]
--output=<Unicode>
```

```
    Overwrite base name use for output files.
                Supports pattern replacements '{notebook_name}'.
    Default: '{notebook_name}'
    Equivalent to: [--NbConvertApp.output_base]
--output-dir=<Unicode>
    Directory to write output(s) to. Defaults
                                  to output to the directory of each notebook.
To recover
                                  previous default behaviour (outputting to the
current
                                  working directory) use . as the flag value.
    Default: ''
    Equivalent to: [--FilesWriter.build_directory]
--reveal-prefix=<Unicode>
    The URL prefix for reveal.js (version 3.x).
            This defaults to the reveal CDN, but can be any url pointing to a
copy
            of reveal.js.
            For speaker notes to work, this must be a relative path to a local
            copy of reveal.js: e.g., "reveal.js".
            If a relative path is given, it must be a subdirectory of the
            current directory (from which the server is run).
            See the usage documentation
            (https://nbconvert.readthedocs.io/en/latest/usage.html#reveal-js-
html-slideshow)
            for more details.
    Default: ''
    Equivalent to: [--SlidesExporter.reveal_url_prefix]
--nbformat=<Enum>
    The nbformat version to write.
            Use this to downgrade notebooks.
    Choices: any of [1, 2, 3, 4]
    Default: 4
    Equivalent to: [--NotebookExporter.nbformat_version]

Examples
--------

    The simplest way to use nbconvert is

            > jupyter nbconvert mynotebook.ipynb --to html

            Options include ['asciidoc', 'custom', 'html', 'latex', 'markdown',
'notebook', 'pdf', 'python', 'qtpdf', 'qtpng', 'rst', 'script', 'slides',
'webpdf'].

            > jupyter nbconvert --to latex mynotebook.ipynb
            Both HTML and LaTeX support multiple output templates. LaTeX
includes

            'base', 'article' and 'report'.  HTML includes 'basic', 'lab' and
            'classic'. You can specify the flavor of the format used.

            > jupyter nbconvert --to html --template lab mynotebook.ipynb

            You can also pipe the output to stdout, rather than a file

            > jupyter nbconvert mynotebook.ipynb --stdout

            PDF is generated via latex

            > jupyter nbconvert mynotebook.ipynb --to pdf

            You can get (and serve) a Reveal.js-powered slideshow

            > jupyter nbconvert myslides.ipynb --to slides --post serve

            Multiple notebooks can be given at the command line in a couple of
            different ways:

            > jupyter nbconvert notebook*.ipynb
            > jupyter nbconvert notebook1.ipynb notebook2.ipynb

            or you can specify the notebooks list in a config file, containing::

                c.NbConvertApp.notebooks = ["my_notebook.ipynb"]

            > jupyter nbconvert --config mycfg.py

To see all available configurables, use `--help-all`.
```

[49]:
```python
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```