Heart Failure Prediction report

Abstract

The project aimed to predict heart failure using the Heart Failure Prediction dataset from Kaggle. Through machine learning techniques, Decision Tree emerged as the most effective model compared to Logistic Regression and K-Nearest Neighbour in forecasting heart failure.

Introduction

The endeavour to forecast heart failure is not only intriguing but also vital due to its potential in healthcare applications. The dataset's significance lies in its comprehensive nature, providing various parameters that could be indicative of heart health. However, there are potential challenges within the dataset, such as skewed distributions or missing values, which may impact the accuracy of predictive models. Furthermore, existing investigations on this dataset might have limitations, prompting a need for a deeper analysis to uncover nuanced patterns and insights for more accurate predictions.

Background

Decision trees function by iteratively splitting the dataset into subsets based on the most significant attributes, creating a tree-like structure of decision nodes. At each node, the algorithm selects the best feature to divide the data, optimising criteria like information gain or Gini impurity. This approach facilitates easy interpretation but can be prone to overfitting.

The logistic regression model operates on a principle of probability, aiming to predict binary outcomes by fitting a logistic curve to the data. It estimates the probability of a particular event occurring based on input features. This model is effective in cases where the dependent variable is categorical.

K-nearest neighbours (KNN) determines the classification of a sample by analysing the majority class of its k nearest neighbours. This algorithm measures distance (commonly using Euclidean distance) to identify the nearest neighbours. It's simple and intuitive but sensitive to noisy data and computationally intensive as it requires calculating distances for each prediction.

Methodology

The dataset was thoroughly examined as part of the technique used for this study. A number of crucial actions were done to guarantee a solid analysis.

1. **Exploratory Data Analysis (EDA):**
    a. To fully comprehend the dataset, a thorough EDA was carried out. This required showing relationships between data, and analysing feature distribution.
        i. The main analysis were between the categorical features vs heart disease and numerical features vs heart disease
2. **Data Preprocessing:**
    a. Handling missing values was an important step. Luckily, there were no missing records in the dataset, making a smoother analysis possible without imputation or deletion.

b. To facilitate their use in the models chosen for analysis, categorical variables were converted into dummy variables.

```
In [23]: df = pd.get_dummies(df, drop_first=True)
```

3. Train test split

```
In [25]: X = df.drop(["HeartDisease"], axis=1)
         y = df["HeartDisease"]

In [26]: X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.15, stratify = y, random_state = 101)
```

This code is preparing the dataset for modelling by splitting it into features (`X`) and the target variable (`y`). The feature set (`X`) excludes the column "HeartDisease," while the target variable (`y`) comprises only the "HeartDisease" column. The dataset is then split into training and testing sets using the `train_test_split` function from Scikit-Learn. It allocates 15% of the data to the test set (`X_test` and `y_test`) and the remaining 85% to the training set (`X_train` and `y_train`). The `stratify` parameter ensures that the class proportions are maintained in both the training and test sets, and `random_state` sets the seed for reproducibility.

4. **Feature Scaling:**
   a. Feature scaling was performed using MinMaxScaler() to standardise the features of the dataset to a similar scale.

```
In [27]: scaler = MinMaxScaler()
         #This scaler is used for feature scaling, speci

In [28]: X_train_scaled = scaler.fit_transform(X_train)
         #This method computes the scaling parameters (l

In [29]: X_test_scaled = scaler.transform(X_test)
         #This method is used to transform the test data
```

   b. Furthermore, an effort was made to use PowerTransform to solve skewness concerns in the data. However, after thorough examination, it was determined that this transformation did not significantly impact model performance.

In [73]:
```
operations = [("scaler", MinMaxScaler()), ("power", PowerTransformer()), ("log", LogisticRegression(random_state=101))]
pipe_log_model = Pipeline(steps=operations)
```

In [35]:
```
pipe_log_model.fit(X_train, y_train)
y_pred = pipe_log_model.predict(X_test)
y_train_pred = pipe_log_model.predict(X_train)

print(confusion_matrix(y_test, y_pred))
print(classification_report(y_test, y_pred))
```

```
[[52 10]
 [ 7 69]]
              precision    recall  f1-score   support

           0       0.88      0.84      0.86        62
           1       0.87      0.91      0.89        76

    accuracy                           0.88       138
   macro avg       0.88      0.87      0.87       138
weighted avg       0.88      0.88      0.88       138
```

After examining the results following skewness adjustment, it's clear that this approach didn't significantly improve our model, particularly when compared to the outcomes achieved by the Logistic Classifier without using PowerTransform. Therefore, for the next stages of this analysis, we'll continue without addressing skewness, as it seems to offer minimal advantages for the results.

5. Assessing Model Accuracy Metrics

In [36]:
```
pipe_scores = cross_validate(pipe_log_model, X_train, y_train, scoring = ['accuracy', 'precision','recall','f1'], cv = 10)
df_pipe_scores = pd.DataFrame(pipe_scores, index = range(1, 11))

df_pipe_scores.head()
```

Out[36]:

	fit_time	score_time	test_accuracy	test_precision	test_recall	test_f1
1	0.139572	0.015473	0.807692	0.791667	0.883721	0.835165
2	0.108036	0.016185	0.820513	0.822222	0.860465	0.840909
3	0.083647	0.008049	0.884615	0.886364	0.906977	0.896552
4	0.108081	0.016376	0.833333	0.875000	0.813953	0.843373
5	0.083779	0.016721	0.897436	0.888889	0.930233	0.909091

In [37]:
```
# Computes the mean values of the performance metrics from the cross-validation
df_pipe_scores.mean()[2:]
```

Out[37]:
```
test_accuracy     0.853846
test_precision    0.857311
test_recall       0.884249
test_f1           0.869920
dtype: float64
```

In [38]:
```
# evaluate a logistic regression model's accuracy
cv = RepeatedStratifiedKFold(n_splits=10, random_state=101)
n_scores = cross_val_score(pipe_log_model, X, y, scoring='accuracy', cv=cv, n_jobs=-1, error_score='raise')
print(f'Accuracy (mean) : {round(n_scores.mean()*100,3)}%, Standard Deviation : {round(n_scores.std()*100,3)}')
```

```
Accuracy (mean) : 85.707%, Standard Deviation : 3.364
```

We created a pipeline that sequentially applies MinMaxScaler, PowerTransformer, and Logistic Regression models, then conducted cross-validation using this pipeline on the training data, and finally stored the evaluation metrics (accuracy, precision, recall, and F1 score) across different folds in a DataFrame for analysis.

These averages represent the model's overall performance across the different folds in the cross-validation, giving an indication of its average predictive ability.

6. **Decision Tree:**

Handles complex feature relationships through tree-like structures. Effective for capturing non-linear patterns in the data.

In [39]:
```
DT_model = DecisionTreeClassifier(class_weight="balanced", random_state=42)
DT_model.fit(X_train_scaled, y_train)
y_pred = DT_model.predict(X_test_scaled)
y_train_pred = DT_model.predict(X_train_scaled)

dt_f1 = f1_score(y_test, y_pred)
dt_acc = accuracy_score(y_test, y_pred)
dt_recall = recall_score(y_test, y_pred)
dt_auc = roc_auc_score(y_test, y_pred)

print(confusion_matrix(y_test, y_pred))
print()
print(classification_report(y_test, y_pred))
print()

confusion_matrix(y_test, y_pred)

train_val(y_train, y_train_pred, y_test, y_pred)
```

```
[[44 18]
 [17 59]]

              precision    recall  f1-score   support

           0       0.72      0.71      0.72        62
           1       0.77      0.78      0.77        76

    accuracy                           0.75       138
   macro avg       0.74      0.74      0.74       138
weighted avg       0.75      0.75      0.75       138
```

Out[39]:

	train_set	test_set
Accuracy	1.0	0.746377
Precision	1.0	0.766234
Recall	1.0	0.776316
f1	1.0	0.771242

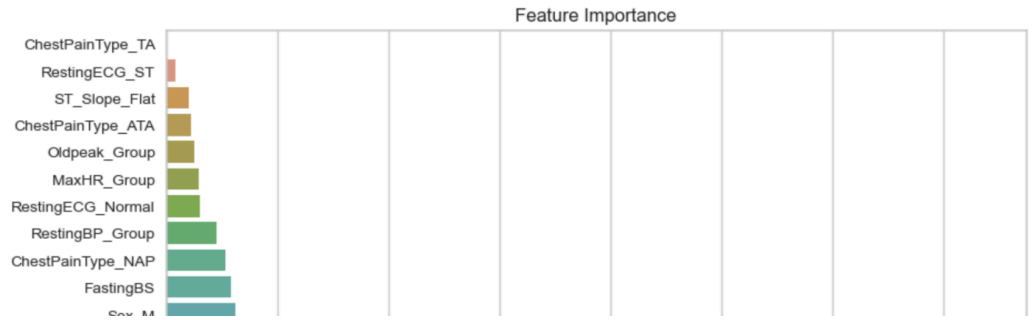
The confusion matrix and classification report for the test set are printed to analyse the classifier's performance on this data. Finally, it calls a function named train_val to assess the model's performance on both train and test sets. This is done for other 2 models as well.

7.1.3) DT feature importance

In [43]:
```
# calculates the scores assigned to each feature by the Decision Tree Classifier.
DT_model.feature_importances_
```

Out[43]:
```
array([0.0804113 , 0.0593962 , 0.04582981, 0.0291426 , 0.10794186,
       0.04572934, 0.02250909, 0.08281583, 0.01471373, 0.0125383 ,
       0.0312367 , 0.01095969, 0.02680743, 0.        , 0.01517614,
       0.00424299, 0.03142039, 0.01015882, 0.36896978])
```

In [44]:
```
# Plots the information above into bar graph
DT_feature_imp = pd.DataFrame(index=X.columns, data = DT_model.feature_importances_,
                              columns = ["Feature Importance"]).sort_values("Feature Importance")
sns.barplot(x=DT_feature_imp["Feature Importance"], y=DT_feature_imp.index)
plt.title("Feature Importance")
plt.show()
```



We're exploring the impact of dropping the most influential feature on our model. Sometimes, an excessively weighted feature can lead to overfitting. To understand its effect, we'll remove this highly influential feature and re-evaluate our model's scores. This process allows us to gauge how much this particular feature influences our model's performance.

In [45]:
```
#This process aims to investigate the model's performance after removing the feature "ST_Slope_Up" from the dataset.
X1 = X.drop(columns = ["ST_Slope_Up"])
y1 = df["HeartDisease"]
```

In [47]:
```
#split new train and test dataset
X1_train, X1_test, y1_train, y1_test = train_test_split(X1, y1, test_size=0.15, random_state=42)
```

In [48]:
```
#Perform the predictions and statistical analysis
operations = [("scaler", MinMaxScaler()), ("dt", DecisionTreeClassifier(class_weight="balanced", random_state=42))]

DT_pipe_model = Pipeline(steps=operations)
DT_pipe_model.get_params()
DT_pipe_model.fit(X1_train, y1_train)

y1_pred = DT_pipe_model.predict(X1_test)
y1_train_pred = DT_pipe_model.predict(X1_train)

print(confusion_matrix(y1_test, y1_pred))
print(classification_report(y1_test, y1_pred))

rf_pipe_f1 = f1_score(y1_test, y1_pred)
rf_pipe_acc = accuracy_score(y1_test, y1_pred)
rf_pipe_recall = recall_score(y1_test, y1_pred)
rf_pipe_auc = roc_auc_score(y1_test, y_pred)

print(confusion_matrix(y1_test, y1_pred))
print()
print(classification_report(y1_test, y1_pred))
print()

confusion_matrix(y_test, y_pred)

train_val(y1_train, y1_train_pred, y1_test, y1_pred)
```

```
[[44 12]
 [18 64]]
              precision    recall  f1-score   support

           0       0.71      0.79      0.75        56
           1       0.84      0.78      0.81        82

    accuracy                           0.78       138
   macro avg       0.78      0.78      0.78       138
weighted avg       0.79      0.78      0.78       138
```

Overall, removing the heavily weighted feature didn't yield any meaningful improvement.

7. **Logistic regression**

Simple, interpretable, and effective in binary classification. Suitable for problems with linearly separable features.

Cross-Validation

Cross-validation techniques were used to guarantee the models' generalizability and dependability. To do this, the dataset was divided into training and validation sets. This made it possible to evaluate the model's performance on various data subsets. This method was essential for determining which model performed the best and for confirming the robustness of the models. This was done for other models as well

7.2.2) Cross-validating LR Model

```
# Conducts cross-validation using Logistic Regression on the training dataset.
log_xvalid_model = LogisticRegression()
log_xvalid_model_scores = cross_validate(log_xvalid_model, X_train_scaled, y_train, scoring = ['accuracy', 'precision','recall',
log_xvalid_model_scores = pd.DataFrame(log_xvalid_model_scores, index = range(1, 11))
log_xvalid_model_scores.mean()[2:]
```

```
test_accuracy     0.857692
test_precision    0.861616
test_recall       0.886575
test_f1           0.873121
dtype: float64
```

```
DT_grid_model.fit(X_train_scaled, y_train)
y_pred = DT_grid_model.predict(X_test_scaled)

y_train_pred = DT_grid_model.predict(X_train_scaled)

dt_grid_f1 = f1_score(y_test, y_pred)
dt_grid_acc = accuracy_score(y_test, y_pred)
dt_grid_recall = recall_score(y_test, y_pred)
dt_grid_auc = roc_auc_score(y_test, y_pred)

print(confusion_matrix(y_test, y_pred))
print()
print(classification_report(y_test, y_pred))
print()

confusion_matrix(y_test,y_pred)

train_val(y_train, y_train_pred, y_test, y_pred)
```

```
Fitting 5 folds for each of 1152 candidates, totalling 5760 fits
[[55  7]
 [ 7 69]]

              precision    recall  f1-score   support

           0       0.89      0.89      0.89        62
           1       0.91      0.91      0.91        76

    accuracy                           0.90       138
   macro avg       0.90      0.90      0.90       138
weighted avg       0.90      0.90      0.90       138
```

7.2.4) Finding the Optimal Threshold Value

In [58]:
```
fp_rate, tp_rate, thresholds = roc_curve(y_test, y_pred_proba[:, 1])
```

In [59]:
```
# Finding the optimal threshold that maximizes the difference between true positive and false positive rates
optimal_idx = np.argmax(tp_rate - fp_rate)
optimal_threshold = thresholds[optimal_idx]
optimal_threshold
```

Out[59]:
```
0.46599508135298984
```

In [60]:
```
# Organizes the ROC curve data into a DataFrame
roc_curve = {"fp_rate":fp_rate, "tp_rate":tp_rate, "thresholds":thresholds}
df_roc_curve = pd.DataFrame(roc_curve)
```

In [61]:
```
# Extracts the values associated with the optimal threshold index
df_roc_curve.iloc[optimal_idx]
```

Out[61]:
```
fp_rate       0.161290
tp_rate       0.934211
thresholds    0.465995
Name: 20, dtype: float64
```

The results indicate that at the identified threshold of approximately 0.466:

The false positive rate is around 0.161.

The true positive rate is notably higher, reaching about 0.934.

These values signify the trade-off between correctly identifying true positives and incorrectly classifying negatives for the chosen threshold. A high true positive rate and a relatively low false positive rate at this threshold suggest a good balance in the model's classification performance.

8. KNN

Non-parametric method for classification tasks. Relies on local relationships between instances in the feature space.

Predictions and statistical analysis without using KNN library

7.3.1) Modelling KNN using Default Parameters

```
# #Perform the predictions and statistical analysis without using KNN library
# Euclidean distance function
def euclidean_distance(p1, p2):
    distance = 0.0
    for i in range(len(p1)):
        distance += (p1[i] - p2[i]) ** 2
    return distance ** 0.5

# KNN algorithm
def knn(train_data, test_data, k):
    predictions = []
    for test_instance in test_data:
        distances = []
        for train_instance in train_data:
            dist = euclidean_distance(train_instance[:-1], test_instance)
            distances.append((train_instance, dist))
        distances.sort(key=lambda x: x[1])
        neighbors = distances[:k]

        # Count the votes for each class
        votes = {}
        for neighbor in neighbors:
            label = neighbor[0][-1]
            if label in votes:
                votes[label] += 1
            else:
                votes[label] = 1

        # Find the majority class
        majority_class = max(votes, key=votes.get)
        predictions.append(majority_class)

    return predictions

# from sklearn.metrics import confusion_matrix, classification_report, f1_score, accuracy_score, recall_score, roc_auc_score

def calculate_metrics(y_true, y_pred):
    f1 = f1_score(y_true, y_pred)
    acc = accuracy_score(y_true, y_pred)
    recall = recall_score(y_true, y_pred)
    auc = roc_auc_score(y_true, y_pred)
    return f1, acc, recall, auc
```

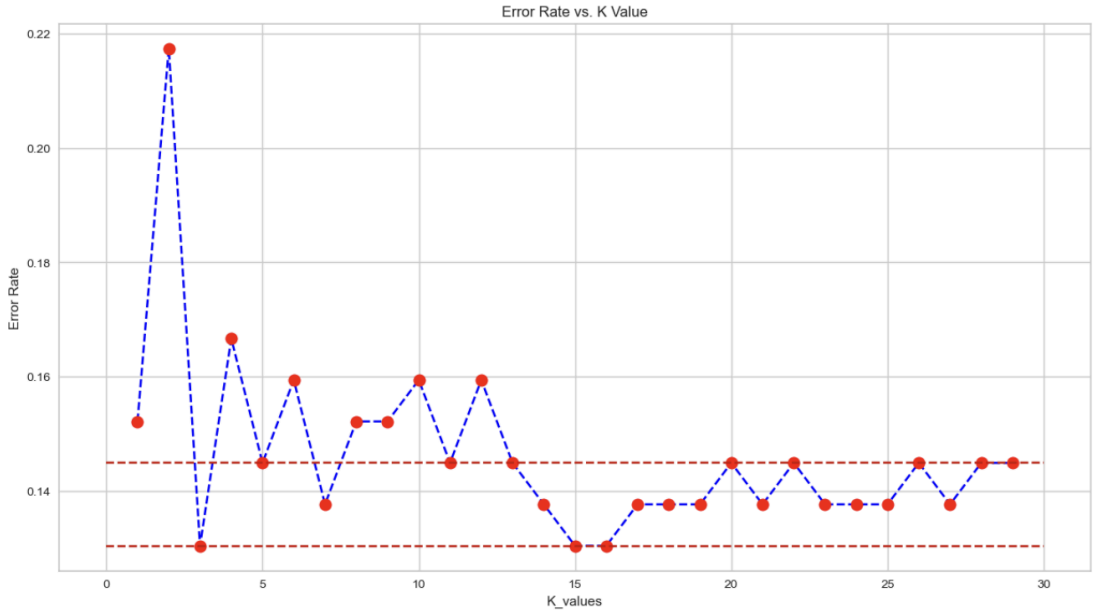
```python
def calculate_metrics(y_true, y_pred):
    f1 = f1_score(y_true, y_pred)
    acc = accuracy_score(y_true, y_pred)
    recall = recall_score(y_true, y_pred)
    auc = roc_auc_score(y_true, y_pred)
    return f1, acc, recall, auc

# Assuming X_train_scaled, y_train, X_test_scaled, y_test are available

# Train the model using your training data
k_value = 5  # Number of neighbors
# Combine features and labels for training data
training_data = []
for features, label in zip(X_train_scaled, y_train):
    combined_data = features + [label]  # Combine features and label
    training_data.append(combined_data)

# Get predictions for the test set
predictions = knn(training_data, X_test_scaled, k_value)

# Get predictions for the training set (for evaluation)
train_predictions = knn(training_data, X_train_scaled, k_value)

# Evaluate the model
# from sklearn.metrics import accuracy_score, roc_auc_score, confusion_matrix, classification_report

knn_acc = accuracy_score(y_test, predictions)
knn_auc = roc_auc_score(y_test, predictions)

# Assuming y_test and y_pred are available
conf_matrix = confusion_matrix(y_test, y_pred)
knn_f1, knn_acc, knn_recall, knn_auc = calculate_metrics(y_test, y_pred)

# Printing confusion matrix and classification report
print("Confusion Matrix:")
print(conf_matrix)
print()
print("Classification Report:")
print(classification_report(y_test, y_pred))
print()

# Perform train_val function
train_val(y_train, y_train_pred, y_test, y_pred)
```

Finding suitable K values

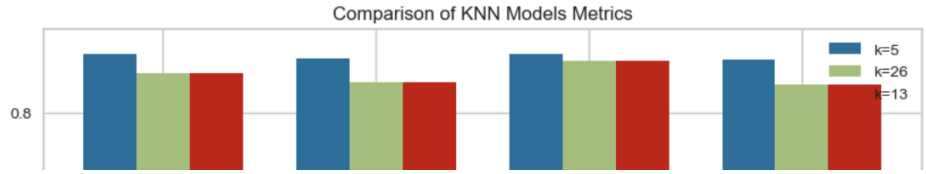
7.3.3) Identifying Suitable K Values Using the Elbow Method

In [65]:
```
# Test error rates
test_error_rates = []

for k in range(1, 30):
    KNN_model = KNeighborsClassifier(n_neighbors=k)
    KNN_model.fit(X_train_scaled, y_train)

    y_test_pred = KNN_model.predict(X_test_scaled)

    test_error = 1 - accuracy_score(y_test, y_test_pred)
    test_error_rates.append(test_error)
```

In [66]:
```
#Plotting k_values with test error rates
plt.figure(figsize=(15, 8))
plt.plot(range(1, 30), test_error_rates, color='blue', linestyle='--', marker='o',
         markerfacecolor='red', markersize=10)
plt.title('Error Rate vs. K Value')
plt.xlabel('K_values')
plt.ylabel('Error Rate')
plt.hlines(y=0.14492753623188404, xmin=0, xmax=30, colors='r', linestyles="--")
plt.hlines(y=0.13043478260869568, xmin=0, xmax=30, colors='r', linestyles="--");
```



In [69]:
```
#Comparing k=5,26 and 13

data = {
    'Metric': ['F1-score', 'Accuracy', 'Recall', 'AUC'],
    'k=5': [knn_f1, knn_acc, knn_recall, knn_auc],
    'k=26': [knn26_f1, knn26_acc, knn26_recall, knn26_auc],
    'k=13': [knn13_f1, knn13_acc, knn13_recall, knn13_auc]  # If available
}

comparison_df = pd.DataFrame(data)

metrics = ['F1-score', 'Accuracy', 'Recall', 'AUC']
k5_metrics = [knn_f1, knn_acc, knn_recall, knn_auc]
k26_metrics = [knn26_f1, knn26_acc, knn26_recall, knn26_auc]
k13_metrics = [knn13_f1, knn13_acc, knn13_recall, knn13_auc]  # If available

bar_width = 0.25
index = range(len(metrics))

plt.bar(index, k5_metrics, bar_width, label='k=5')
plt.bar([i + bar_width for i in index], k26_metrics, bar_width, label='k=26')
plt.bar([i + 2 * bar_width for i in index], k13_metrics, bar_width, label='k=13')  # If available

plt.xlabel('Metrics')
plt.ylabel('Values')
plt.title('Comparison of KNN Models Metrics')
plt.xticks([i + bar_width for i in index], metrics)
plt.legend()
plt.show()
```



k=5 is the best among all the others

Results

These models were chosen due to the heart failure dataset containing both linear and non-linear relationships between health parameters and heart disease presence. The selection aimed to leverage the different strengths of these algorithms to identify the most suitable one for predicting heart disease in this specific context. By following those methodological steps, a comprehensive evaluation of the dataset and models was undertaken to arrive at the conclusion that Decision Tree was the optimal choice for predicting heart failure in this dataset.

In [71]:
```
compare = pd.DataFrame({
    "Model": ["Decision Tree", "KNN", "Logistic Regression"],
    "F1": [dt_grid_f1, knn_f1, log_f1],
    "Recall": [dt_grid_recall, knn_recall, log_recall],
    "Accuracy": [dt_grid_acc, knn_acc, log_acc],
    "ROC_AUC": [dt_grid_auc, knn_auc, log_auc]
})

print(compare)
```

```
                 Model        F1    Recall  Accuracy   ROC_AUC
0        Decision Tree  0.907895  0.907895  0.898551  0.897496
1                  KNN  0.907895  0.907895  0.898551  0.897496
2  Logistic Regression  0.890323  0.907895  0.876812  0.873302
```

Decision Tree and KNN outperformed Logistic Regression in terms of accuracy, precision and F1 score.

Evaluation

1. **Data and Assumptions:**
    a. The dataset exhibited no missing values and was deemed suitable for analysis.
2. **Feature Transformation:**
    a. Transformation of categorical variables into dummies enabled the models to use them effectively.
    b. Skewness handling, while attempted, did not yield noticeable improvements, leading to the decision to proceed without such adjustments.
3. **Model Performance:**
    a. When compared to the Logistic Regression model, Decision Tree and KNN models consistently performed higher in terms of accuracy, precision and F1-score.
4. **Limitations:**
    a. It's possible that some features in the dataset were absent, which would have improved model predictions.
    b. Furthermore, despite efforts, managing skewness did not result in the anticipated enhancements in model performance.
5. **Future Directions:**
    a. Future iterations could explore feature engineering with different techniques or consider gathering additional data to enhance prediction capabilities.
    b. Experimenting with more advanced algorithms might lead to better predictions for heart disease classification.

Conclusions

Throughout this project, the primary objective was to predict heart disease occurrences using various machine learning models. The analysis involved extensive data exploration, model building, and evaluation.

The dataset was meticulously investigated, revealing no missing values and enabling detailed feature analysis and transformation. Categorical variables were transformed into dummy variables to facilitate model compatibility. While attempts to handle skewness didn't significantly enhance model performance, this step was skipped in subsequent analyses.

Three main models—Decision Tree, K-Nearest Neighbors (KNN), and Logistic Regression—were employed, with Decision Tree and KNN consistently displaying high accuracy and recall. KNN's performance was evaluated with different K-values, emphasising the importance of selecting the optimal K for improved results.

Model evaluation illustrated that Decision Tree and KNN displayed almost identical performance across various metrics, showcasing high F1-scores and recall. Logistic Regression, while still performing well, showed slightly lower metrics.

In conclusion, both Decision Tree and KNN proved to be robust models for predicting heart disease, consistently delivering high accuracy and recall. Logistic Regression, while competitive, slightly lagged behind in performance metrics. The evaluation underscores the significance of tuning hyperparameters for better model performance in medical predictive analytics.

References

- Dataset: https://www.kaggle.com/datasets/fedesoriano/heart-failure-prediction
- Load dataset: https://pandas.pydata.org/docs/reference/api/pandas.read_csv.htmlhttps
- Pipeline: https://scikit-learn.org/stable/modules/generated/sklearn.pipeline.Pipeline.html

Note: The notebook (html) has all the explanation of the code and the results