**DURAN DUMAN 190101037**

**Project-3 Report**

**1- Introduction**

My main goal for this project was to create a decision tree classifier that can determine the the presence of hypertension in individuals. The dataset I worked with contains different features that have influence on hypertension presence, including **age**, **gender**, **family history of blood pressure**, **physical activity**, **smoking status**, **alcohol consumption** and **blood pressure disease status**.

**Data Collection**

The aim for the project was to train a model that could predict presence of hypertension based on these features. To construct a robust model, **I gathered a dataset of 135 data points using Google Forms.**

This dataset used as a training set (**traindata.xlsx**) for model training and evaluation set for performance assessment through a **5-fold cross-validation** approach.

The decision tree classifier was chosen as the model of choice for its interpretability and suitability for this classification task.

**1.1 Features and Target Variable**

***Features (X):***

* Age :***( [0-30] , [31-50] , [51-70] ) => (1, 2, 3)***
* Gender : ***(Male, Female) => (1, 2)***
* Family History of Hypertension : ***(There is, There isn’t) => (1, 2)***
* Physical Activity : ***(Low, Medium, High) => (1, 2, 3)***
* Smoking Status : ***(I smoke, I don’t smoke) => (1, 2)***
* Alchocol Consumption : ***(I drink, I don’t drink) => (1, 2)***

***Target Variable (y):***

* Hypertension Status : ***(I have, I haven’t) => (1,2)***

The features chosen for this decision tree classifier were carefully selected based on their known influence on hypertension. Each feature provides valuable information that contributes to the model's ability to predict the presence of hypertension in individuals. Here's a brief overview of why these specific features were included:

* **Age:** Age is a fundamental risk factor for hypertension, reflecting the increased likelihood of developing high blood pressure as individuals age.
* **Gender:** Hypertension prevalence can differ between genders, making gender an important factor in understanding and predicting blood pressure issues.
* **Family History of Hypertension:** Genetic predisposition plays a crucial role in hypertension, making a family history of blood pressure a significant predictor.
* **Physical Activity:** Regular exercise is associated with cardiovascular health, and the lack of physical activity is a known risk factor for hypertension.
* **Smoking Status:** Smoking has a direct impact on blood pressure, and individuals who smoke are at a higher risk of developing hypertension.
* **Alcohol Consumption:** Excessive alcohol consumption is linked to hypertension, and including this feature allows the model to account for its influence on blood pressure.
* **Hypertension Status:** This feature provides direct information about an individual's current blood pressure condition, serving as a strong predictor for hypertension.

By incorporating these features, the decision tree model aims to capture the multifaceted nature of hypertension, considering factors related to age, genetics, lifestyle, and current health status. This comprehensive approach enhances the model's ability to make accurate predictions during training and cross-validation.

**1.2 Libraries Used**

I used two main Python libraries for this project:

* **Pandas**: I used Pandas for data manipulation and analysis. Pandas helped me read the dataset from excel files into DataFrames, making it easy to preprocess the data.
* **Numpy**: I used Numpy especially in the calculation of entropy and information gain within the decision tree algorithm.
* **StratifiedKFold (from sklearn.model\_selection):** Used for implementing stratified k-fold cross-validation, ensuring balanced class distribution across folds.

import pandas as pd  
import numpy as np

from sklearn.model\_selection import StratifiedKFold

**2- Data Reading and Preparation:**

To begin, I read the dataset from excel files into pandas DataFrames. I specified column names and then separated the features (X) from the target variable (y) for the training set.

# Veri setini okuma işlemi  
col\_names = ["Cinsiyet", "Yaş", "AiledeTansiyon", "FizikselAktivite", "Sigaraİçme", "AlkolTüketimi", "Tansiyon"]  
train\_data = pd.read\_excel(r"traindata.xlsx", names=col\_names)  
  
X\_train = train\_data.drop("Tansiyon", axis=1)  
y\_train = train\_data["Tansiyon"]

This separation was a must to enable the model to learn from the features and predict the target variable accurately.

**3- Decision Tree Implementation Details**

In this section, I showed a thorough breakdown of each function in the decision tree implementation.

***3.1 entropy\_hesapla***

# Veri kümesinin entropisini hesaplayan fonksiyon  
def entropy\_hesapla(y):

# kaç farklı class olduğunu ve bu classlardan kaç tane olduğunu hesaplar  
 unique\_classes, class\_counts = np.unique(y, return\_counts=True)

# her classın veri kümesindeki oranını hesaplar  
 probabilities = class\_counts / len(y)

# entropi hesaplar  
 entropy\_value = -np.sum(probabilities \* np.log2(probabilities))   
 return entropy\_value

**Purpose**: Calculates the entropy of a given set of labels (y), representing the uncertainty in the set.

**Process:**

* **unique\_classes, class\_counts = np.unique(y, return\_counts=True):** Calculates the unique classes and their corresponding counts in the input set.
* **probabilities = class\_counts / len(y):** Computes the probabilities of each class.
* **entropy\_value = -np.sum(probabilities \* np.log2(probabilities**)): Calculates the entropy using the standard formula for entropy.

**Output:** Returns the computed entropy value.

***3.2 information\_gain\_hesapla***

# Belirli bir feature için information gain hesaplayan fonksiyon  
def information\_gain\_hesapla(X, y, feature):

# mevcut durumun entropisini hesaplar  
 entropy\_before = entropy\_hesapla(y)

# feature'daki unique değerleri alır  
 unique\_values = np.unique(X[feature])  
 information\_after = 0  
  
 for value in unique\_values: # her bir unique değer için

# feature'ın bu değere sahip olduğu alt kümesi  
 subset\_y = y[X[feature] == value]

# information'ı hesaplar  
 information\_after += len(subset\_y) / len(y) \*

entropy\_hesapla(subset\_y)

# information gain'i hesaplar  
 information\_gain\_value = entropy\_before - information\_after   
 return information\_gain\_value

**Purpose**: Computes the information gain for a specific feature in the dataset.

**Process:**

* **entropy\_before = entropy\_hesapla(y):** Computes the entropy of the dataset before the split.
* **unique\_values = np.unique(X[feature]):** Finds unique values for the specified feature.
* **information \_after = 0**: Initialize the variable to store the information after the split
* ***For each unique value of the feature:***
  + **subset\_y = y[X[feature] == value]:** Selects the subset of target variable values for the current feature value.
  + **information \_after += len(subset\_y) / len(y) \* entropy\_hesapla(subset\_y):** Calculates the information after the split.
* **information\_gain\_value = entropy\_before - information \_after:** Computes the information gain.

**Output:** Returns the information gain value for the specified feature.

***3.3 find\_best\_split***

# En iyi bölünme noktasını bulan fonksiyon (Gain'e bakarak)  
def find\_best\_split(X, y):

best\_feature = None # en iyi feature için başlangıç değeri

# en iyi information gain için başlangıç değeri  
 best\_information\_gain = 0

# verilen her bir feature için information gain hesaplar  
 for feature in X.columns:

current\_information\_gain = information\_gain\_hesapla(X, y, feature)

#current information gain, şu ana kadar en iyisiyse best feature'ı

güncelle  
 if current\_information\_gain > best\_information\_gain:   
 best\_information\_gain = current\_information\_gain  
 best\_feature = feature  
  
 return best\_feature # best feature'ı döndürür

**Purpose:** It Identifies the best feature to split on based on the maximum information gain.

**Process:**

* **best\_feature = None:** Initializes the best feature as None.
* **best\_information\_gain = 0:** Initializes the best information gain as 0.
* ***For each feature in the feature matrix:***
  + **current\_information\_gain = information\_gain\_hesapla(X, y, feature):** Calculates the information gain for the current feature.
  + Updates **best\_feature** and **best\_information**\_gain if the current information gain is higher.
* **Output: Returns the best feature for splitting.**

***3.4 id3\_build\_tree***

# ID3 Algoritmasını kullanarak Decision Tree (Karar Ağacı) oluşturan fonksiyon.  
def id3\_build\_tree(X, y):

# eğer veri kümesinde yalnızca bir class varsa, bir leaf node oluştur

ve class'ı döndür  
 if len(np.unique(y)) == 1:   
 return {'class': y.iloc[0]}

# best feature'u bulur (gain'e göre)

best\_feature = find\_best\_split(X, y)

# eğer best feature'u yoksa, bir leaf node oluştur ve en çok tekrar

eden class'ı döndür  
 if best\_feature is None:  
 return {'class': np.argmax(np.bincount(y))}  
  
 tree = {'feature': best\_feature, 'branches': {}} # ağacı oluştur

# best feature'un her bir unique değeri için sub tree oluşturur  
 for value in np.unique(X[best\_feature]):  
 subset\_X = X[X[best\_feature] == value].drop(best\_feature, axis=1)  
 subset\_y = y[X[best\_feature] == value]

# sub tree oluşturur ve ana tree'nin alt dallarına ekler (recursive

devam eder işlemler)  
 tree['branches'][value] = id3\_build\_tree(subset\_X, subset\_y)  
  
 return tree

**Process:**

* If all samples have the same label, a leaf node is created with the class label.
* **best\_feature = find\_best\_split(X, y):** Finds the best feature for splitting.
* If no suitable feature is found for splitting, a leaf node is created with the most frequent class label in the current subset.
* Initializes a tree dictionary with the best feature and an empty 'branches' dictionary.
* *For each unique value of the best feature:*
* Subset the data where the selected feature has the current unique value.
* Recursively call id3\_build\_tree on the subset of data for that feature value.
* Populate the 'branches' dictionary with the feature value as the key and the recursively constructed subtree as the value.

**Output**: Returns the constructed decision tree.

***3.5 predict***

# Karar ağacını kullanarak bir test data için tahmin yapar.  
def predict(tree, sample):

# eğer bu düğüm bir leaf node ise, class'ı döndür  
 if 'class' in tree:   
 return tree['class']  
  
 feature\_value = sample[tree['feature']]

# feature'ın değeri ağaçta yoksa, "None" döndür  
 if feature\_value not in tree['branches']:   
 return None  
  
 branch = tree['branches'][feature\_value]

# tahminler ağacın geri kalanı için recursive şekilde devam eder  
 return predict(branch, sample)

**Process:**

* If the current node is a leaf node, returns the class label.
* Checks the value of the feature in the current node of the tree.
* If the feature value is not in the branches, returns “None” indicating that the algorithm cannot make a prediction for this specific path in the tree.
* Recursively calls predict with the subtree corresponding to the feature value.

**Output:** Returns the predicted class label for the input sample.

**3.6 *print\_tree***

def print\_tree(tree, indent=0):  
 if "class" in tree:  
 print("|\_\_ Sınıf:", tree["class"])  
 else:  
 print("|\_\_ Özellik:", tree["feature"])  
 for value, subtree in tree["branches"].items():  
 print("| " \* indent + "|\_\_ {}: ".format(value), end="")  
 print\_tree(subtree, indent + 1)

**Process:**

* If the current node is a leaf node, prints the class label.
* If the current node is a decision node, prints the feature and iterates through branches.
* Recursively calls print\_tree for each branch with increased indentation.

**Output:** Prints the structure of the decision tree.

**3.7 *calculate\_confusion\_matrix\_manual***

# Confiusion Matrix'i hesaplayan fonksiyon.  
def calculate\_confusion\_matrix\_manual(y\_true, y\_pred):

# true ve predicted y değerleri birleşiminden unique classları elde

eder.  
 unique\_classes = np.unique(np.concatenate((y\_true, y\_pred)))

num\_classes = len(unique\_classes) # Toplam class sayısını al.

# Confusion matrixi sıfırlarla başlat.  
 confusion\_matrix = np.zeros((num\_classes, num\_classes), dtype=int)

# Her bir true ve predicted değer confusion matrixi güncelle.  
 for true\_label, predicted\_label in zip(y\_true, y\_pred):

# True ve predicted sınıfların indekslerini belirleme

true\_idx = np.where(unique\_classes == true\_label)[0][0]  
 predicted\_idx = np.where(unique\_classes == predicted\_label)[0][0]

# Confusion matrix'i günceller.  
 confusion\_matrix[true\_idx, predicted\_idx] += 1

return confusion\_matrix # Hesaplanan confusion matrix'i döndür.

**Process:**

• Obtaining unique classes by concatenating true and predicted values.

• Determining the total number of classes.

• Initializing a confusion matrix with zeros.

• Updating the confusion matrix for each true and predicted pair.

**Output:** returns the computed confusion matrix. It is a 2D array showing the counts of true positive, true negative, false positive, and false negative instances.

"""  
[[True Negative (TN) False Positive (FP)]  
 [False Negative (FN) True Positive (TP)]]  
"""

Above matrix shows the structure of a returned confusion matrix including “True Positive”, “True Negative”, “False Positive”, “False Negative”.

**3.8 *calculate\_f1\_score\_manual***

# F1 skoru hesaplama fonksiyonu  
def calculate\_f1\_score\_manual(confusion\_matrix):  
 tn, fp, fn, tp = confusion\_matrix.ravel()  
  
 precision = tp / (tp + fp) if (tp + fp) != 0 else 0  
 recall = tp / (tp + fn) if (tp + fn) != 0 else 0  
  
 f1\_score\_manual = 2 \* (precision \* recall) / (precision + recall) if (precision + recall) != 0 else 0  
  
 return f1\_score\_manual

**Process:**

* The function takes a confusion matrix as input, which is a 2x2 matrix containing the counts of true negatives (tn), false positives (fp), false negatives (fn), and true positives (tp).
* Unpacks the values from the confusion matrix into variables tn, fp, fn, and tp.
* Calculates precision: Precision is the ratio of true positives to the sum of true positives and false positives. **(tp / (tp + fn) if tp + fn != 0, otherwise set as 0.**
* Calculates recall: Recall is the ratio of true positives to the sum of true positives and false negatives. **(tp / (tp + fn) if tp + fn != 0, otherwise set as 0.**
* Calculates F1 score: **2 x (precision x recall) / (precision + recall)**
* Returns the calculated F1 score.

**Output:** The function outputs the F1 score, a metric that combines both precision and recall, providing a single value that represents the model's performance.

***3.9 k\_fold\_cross\_validation***

# K-fold cross validation ve confusion matrix hesaplama.  
def k\_fold\_cross\_validation(X, y, k=5):  
 skf = StratifiedKFold(n\_splits=k, shuffle=True, random\_state=42) # veriyi 5 parçaya böler, 5 fold sağlar.  
 fold = 1 #fold sayacı  
  
 # Her fold için confusion matrix ve F1 skorlarını saklamak için listeler.  
 confusion\_matrices = []  
 f1\_scores\_manual = []  
 accuracies=[]  
  
 # K-Fold döngüsü  
 for train\_index, test\_index in skf.split(X, y):  
 print(f"\nFold {fold}:")  
  
 # Train ve test setlerini oluşturur.  
 X\_train\_fold, X\_test\_fold = X.iloc[train\_index], X.iloc[test\_index]  
 y\_train\_fold, y\_test\_fold = y.iloc[train\_index], y.iloc[test\_index]  
  
 # Karar ağacını oluşturur.  
 tree = build\_tree(X\_train\_fold, y\_train\_fold)  
  
 # Test seti için tahminleri yapar.  
 predictions = X\_test\_fold.apply(lambda x: predict(tree, x), axis=1)  
  
 # X\_test\_fold'ları, gerçek değerlerini ve tahminleri Excel

dosyalarına kaydetme  
 result\_df = pd.concat([X\_test\_fold, pd.DataFrame({'True\_Label':

y\_test\_fold, 'Predictions': predictions})],axis=1)

result\_df.to\_excel(f'fold\_{fold}\_testData.xlsx', index=False)

# Confusion matrix hesaplama  
 confusion = calculate\_confusion\_matrix\_manual(y\_test\_fold,

predictions)  
 confusion\_matrices.append(confusion)  
  
 # Accuracy hesaplama  
 accuracy = np.trace(confusion) / np.sum(confusion)  
 accuracies.append(accuracy)  
  
 # F1 skoru hesaplama  
 f1\_manual = calculate\_f1\_score\_manual(confusion)  
 f1\_scores\_manual.append(f1\_manual)  
  
 print("Confusion Matrix:")  
 print(confusion)  
  
 # TN, TP, FP, FN değerlerini bulup print eder  
 tn, fp, fn, tp = confusion.ravel()  
 print(f"True Negative (TN): {tn}, True Positive (TP): {tp}, False

Positive (FP): {fp}, False Negative (FN): {fn}")

print(f"Accuracy: {accuracy:.4f}")  
 print(f"F1 Score: {f1\_manual:.4f}")  
  
 fold += 1  
  
 return confusion\_matrices, f1\_scores\_manual

**Process:**

* **Stratified K-Fold Splitting:** The data is split **into 5 folds using Stratified K-Fold**, ensuring that each fold maintains the same class distribution as the original dataset. This is crucial to avoid biased training and testing sets.
* **Fold Iteration:** The function iterates through each fold, creating distinct training and testing sets for every iteration.
* **Decision Tree Construction:** For each fold, a decision tree is constructed using the training set. The build\_tree function, is responsible for creating the decision tree.
* **Prediction:** The constructed decision tree is then used to make predictions on the test set. The predictions are stored in the predictions variable using the predict function.
* **Data Saving:** The function saves the X\_test\_fold (test data), true values and predicstions into an Excel file named 'fold\_{fold}\_testData.xlsx' for each fold. This allows for later inspection of the test data used in each fold.
* **Confusion Matrix Calculation:** The function calculates the confusion matrix using the calculate\_confusion\_matrix\_manual function. The confusion matrix provides a detailed breakdown of the model's performance, showing the counts of true positives, true negatives, false positives, and false negatives.
* **Accuracy Computation:** The accuracy of the model for the current fold is computed by dividing the sum of the diagonal elements of the confusion matrix (true positives and true negatives) by the total number of instances.
* **F1 Score Calculation:** The F1 score is computed using the calculate\_f1\_score\_manual function. It combines precision and recall to provide a single metric that balances both false positives and false negatives.
* **Results Printing**: The confusion matrix, TN, TP, FP, FN values, accuracy, and F1 score are printed for each fold, providing insights into the model's performance across different subsets of the data.

**Output:**

The **k\_fold\_cross\_validation** function returns two lists: **confusion\_matrices** and **f1\_scores**.

* **confusion\_matrices:** A list containing the confusion matrices for each fold. Each element of the list represents the confusion matrix of a specific fold.
* **f1\_scores:** A list containing the F1 scores for each fold. Each element of the list represents the F1 score of a specific fold.

These lists provide a evaluation of the decision tree classifier's performance across different folds.

***3.10 calculate\_average\_f1\_score***

# Overall F1 skoru hesaplama  
def calculate\_average\_f1\_score(f1\_scores):  
 # F1 skorlarının toplamını alıp ve fold sayısına bölerek ortalamayı hesaplar.  
 overall\_f1\_score = sum(f1\_scores) / len(f1\_scores)  
 print("\nOverall F1 Score:", overall\_f1\_score)

**Process:**

• Sum up the F1 scores obtained for each fold.

• Divide the total sum by the number of folds to calculate the average.

**Output:**

Prints the calculated overall F1 score.

***3.11 calculate\_average\_metrics***

# K-fold sonuçlarından genel doğruluk ve ortalama confusion matrix hesaplayan fonksiyon  
def calculate\_average\_metrics(confusion\_matrices):  
 # doğruluk ve confusion matrixlerin ortalaması için başlangıç değerleri  
 overall\_accuracy = 0  
 average\_confusion\_matrix = np.zeros\_like(confusion\_matrices[0], dtype=float)  
  
 # Her bir fold için döngü  
 for i, confusion\_matrix in enumerate(confusion\_matrices):  
  
 # Confusion Matrix üstünden her fold için doğruluk hesaplar ve print eder.  
 accuracy = np.trace(confusion\_matrix) / np.sum(confusion\_matrix) # (True Positive + True Negative) / sum(Confusion Matrix) işlemi.  
 overall\_accuracy += accuracy  
  
 # Average confusion matrix'e her fold'un confusion matrix'ini ekler  
 average\_confusion\_matrix += confusion\_matrix  
  
 #ortalama accuracy değeri hesaplanır  
 overall\_accuracy /= len(confusion\_matrices)  
  
 # Average confusion matrix'i fold sayısına böler. Böylece average confusion matrix elde edilir.  
 average\_confusion\_matrix /= len(confusion\_matrices)  
  
 # Elde edilen değerleri ekrana yazdırma işlemi.  
 print("\nOverall:")  
 print(f"Overall Accuracy: {overall\_accuracy:.4f}")  
 print("Average Confusion Matrix:")  
 print(average\_confusion\_matrix)  
  
 # TN, TP, FP, FN değerlerini bulup print eder  
 tn, fp, fn, tp = average\_confusion\_matrix.ravel()  
 print(f"True Negative (TN): {tn}, True Positive (TP): {tp}, False Positive (FP): {fp}, False Negative (FN): {fn}")

**Process:**

* Initializes overall accuracy and an average confusion matrix.
* Iterates through each fold, calculates accuracy, and updates overall accuracy and the average confusion matrix.
* Computes the average accuracy and confusion matrix by dividing by the number of folds.
* Outputs the overall accuracy, average confusion matrix, and individual values (TN, TP, FP, FN) derived from the average confusion matrix.

**Output:** Prints,

* Overall accuracy, representing the average accuracy across all folds.
* Average confusion matrix, offering a consolidated view of model performance.
* Individual values (TN, TP, FP, FN) for a detailed assessment of the model's predictive performance.

***3.12 Handling Functions***

# K-fold cross validation ve confusion matrixleri hesaplama  
confusion\_matrices, f1\_scores = k\_fold\_cross\_validation(X\_train, y\_train)  
  
# Toplam accuracy ve ortalama confusion matrixi hesaplama  
calculate\_average\_metrics(confusion\_matrices)  
  
# Toplam F1 skoru hesaplama  
calculate\_average\_f1\_score(f1\_scores\_manual\_f1)

**K-fold Cross Validation with Confusion Matrices and F1 Scores:**

This line executes k-fold cross-validation on the training data (X\_train, y\_train), yielding confusion matrices and F1 scores for each fold.

**Calculate Average Metrics (Accuracy and Confusion Matrix):**

Invokes a function (calculate\_average\_metrics) to compute overall accuracy and average confusion matrix from the list of confusion matrices obtained during k-fold cross-validation.

**Calculate Average F1 Score:**

Calls a function (calculate\_average\_f1\_score) to determine the average F1 score from the list of F1 scores (f1\_scores\_manual\_f1) obtained during k-fold cross-validation. The variable name suggests it might be manually calculated F1 scores.

**4 - Confusion Matrix, F1-Score and Accuracy Evaluation Based on Folds**

Fold 1:  
Confusion Matrix:  
[[6 6]  
 [7 8]]  
True Negative (TN): 6, True Positive (TP): 8, False Positive (FP): 6, False Negative (FN): 7  
Accuracy: 0.5185  
F1 Score: 0.5517  
  
Fold 2:  
Confusion Matrix:  
[[8 4]  
 [7 8]]  
True Negative (TN): 8, True Positive (TP): 8, False Positive (FP): 4, False Negative (FN): 7  
Accuracy: 0.5926  
F1 Score: 0.5926  
  
Fold 3:  
Confusion Matrix:  
[[9 2]  
 [7 9]]  
True Negative (TN): 9, True Positive (TP): 9, False Positive (FP): 2, False Negative (FN): 7  
Accuracy: 0.6667  
F1 Score: 0.6667  
  
Fold 4:  
Confusion Matrix:  
[[6 5]  
 [7 9]]  
True Negative (TN): 6, True Positive (TP): 9, False Positive (FP): 5, False Negative (FN): 7  
Accuracy: 0.5556  
F1 Score: 0.6000  
  
Fold 5:  
Confusion Matrix:  
[[6 5]  
 [7 9]]  
True Negative (TN): 6, True Positive (TP): 9, False Positive (FP): 5, False Negative (FN): 7  
Accuracy: 0.5556  
F1 Score: 0.6000

In In the provided output, each fold's confusion matrix are presented:

* **Fold 1’s Confusion Matrix:**

True Negative (TP): 6, True Positive (TN): 8, False Positive (FP): 6, False Negative (FN): 7

* **Fold 2’s Confusion Matrix:**

True Negative (TP): 8, True Positive (TN): 8, False Positive (FP): 4, False Negative (FN): 7

* **Fold 3’s Confusion Matrix:**

True Negative (TP): 9, True Positive (TN): 9, False Positive (FP): 2, False Negative (FN): 7

* **Fold 4’s Confusion Matrix:**

True Negative (TP): 6, True Positive (TN): 9, False Positive (FP): 5, False Negative (FN): 7

* **Fold 5’s Confusion Matrix:**

True Negative (TP): 6, True Positive (TN): 9, False Positive (FP): 5, False Negative (FN): 7

The confusion matrix provides detailed information on the model's predictions, distinguishing between true positives, true negatives, false positives, and false negatives.

In this provided output, each fold's accuracy values and f1 scores are presented:

* **Fold 1’s Accuracy and F1 Score: Accuracy: 0.5185, F1 Score: 0.5517**
* **Fold 2’s Accuracy and F1 Score: Accuracy: 0.5926, F1 Score: 0.5926**
* **Fold 3’s Accuracy and F1 Score: Accuracy: 0.6667 ,F1 Score: 0.6667**
* **Fold 4’s Accuracy and F1 Score: Accuracy: 0.5556, F1 Score: 0.6000**
* **Fold 5’s Accuracy and F1 Score: : Accuracy: 0.5556, F1 Score: 0.6000**

The F1 score is a combined metric of precision and recall, further emphasizes the model's effectiveness on each fold.

These accuracy values and f1 scores represent the overall correctness of the model predictions for each fold. The combination of confusion matrices, accuracy and f1 score provides a comprehensive understanding of the model's performance across different subsets of the dataset during k-fold cross-validation.

**You can see the each fold’s prediction excel files in the Project file.**

***5. Overall Evaluation and Results***

Overall:  
Overall Accuracy: 0.5778  
Average Confusion Matrix:  
[[7. 4.4]  
 [7. 8.6]]  
True Negative (TN): 7.0, True Positive (TP): 8.6, False Positive (FP): 4.4, False Negative (FN): 7.0  
Overall F1 Score: 0.6021966794380587

The overall evaluation provides a summary of the decision tree classifier's performance across all folds.

* **Overall Accuracy:**

The overall accuracy is calculated by averaging the accuracy values obtained from each fold. In this case, the overall accuracy is **0.5778 (57.78%),** indicating the proportion of correct predictions across all folds.

* **Average Confusion Matrix:**

The average confusion matrix is obtained by taking the mean of the individual confusion matrices from each fold.

It gives an overall picture of the model's performance, including values for **true positives, true negatives, false positives, and false negatives**. In this case, the average confusion matrix is represented as:

[[7. 4.4]  
 [7. 8.6]]

* **Overall F1 Score:**

The overall F1 score is computed by averaging the F1 scores obtained for each fold. The calculated overall F1 score is 0.6022 **(60.22%)**, reflecting the model's effectiveness in terms of both false positives and false negatives.

|  |  |
| --- | --- |
| **Overall Accuracy** | **57.78%** |
| **Overall F1 Score** | **60.22%** |

The review of the decision tree classifier's performance across all the sets of data provides important information about how well it predicts the presence of hypertension. The accuracy of **57.78%** shows that the model is quite good at making correct predictions on different parts of the dataset.

Looking at the average confusion matrix gives us a more detailed picture of how the classifier behaves. It's good at identifying **true positives (8.6) and true negatives (7.0)**, which are cases where it got the prediction right. However, it sometimes predicts **false positives (4.4) and false negatives (7.0)**, indicating areas where the model could be improved.

The overall F1 score of **60.22%**, tells us that the model does well in finding a balance between being precise and recalling instances. In predicting hypertension, it's important to minimize both false positives and false negatives, and this score shows that the model is doing reasonably well.