Running Performance Classification Project

1. Problem Statement

- **Objective:** Classify runners into performance tiers (Beginner / Intermediate / Advanced) based on running metrics. Compare between Clustering to true labels.

---

2. Data Collection

- Based on friends' running data (Garmin Connect, RunKeeper, Apple Watch, Nike run club, Strava).
- Kaggle - "Running Log Insight," "Running races from Strava," "strava-data."
- **Limitations:** Small datasets (~280 runners overall).

---

3. Method

- Gather the datasets and create a median run for each runner, then use the chosen running metric and run the model.
- Using Riegel formula - https://trainasone.com/ufaq/riegels-formula/ to create a custom running metric.
- Splitting the data to train and test by using `traintestsplit` function (splits randomly the data to train split and test split, sizes can be determine by the user) and Cross Validation (divides the data to multiple 'folds', every time using different 'fold' as the test split and running the model than aggregate the model results and returns their average).
- **Classification models:** Decision Tree, Logistic Regression, Random Forest, Gradient Boosting.
- **Clustering model:** K-means.

---

4. Measuring Model Performance

- **Common classification metrics:**
  - **Accuracy:** Measures the ratio of True classifications. Formula: $(TP + TN)/(TP + TN + FP + FN)$
  - **Precision:** Measures the ratio of True Positive out of all the positive classifications in the model. Formula: $TP/(TP + FP)$
  - **Recall:** Measures the ratio of True Positive out of all real positives. Formula: $TP/(TP + FN)$

    - **F1 Score:** Harmonic mean of its precision and recall, providing a single value that balances both. Formula: $2 * (Precision * Recall) / (Precision + Recall)$
- **Decision Tree metrics:**
    - **Gini:** Measures the impurity of data within a decision tree node. When 0, the node is pure (contains only elements of a single class). Higher values mean the node is impure (has a more mixed distribution of classes).
- **Classification Summaries:**
    - **Support:** Number of actual occurrences of the class in the specified dataset.
    - **Macro Avg:** The unweighted average of the per-class metrics. It calculates the metric for each class and then takes a simple arithmetic mean.
    - **Weighted Avg:** The average of the per-class metrics, weighted by the support for each class.
- **Common Clustering metrics:**
    - **Internal Clustering Validation Metrics:**
        - **Inertia:** The sum of the squared distances between each data point and the centroid of its assigned cluster. It measures how 'tight' or 'compact' the cluster is. A lower inertia value indicates that the data points are closer to their cluster centroids, meaning the clusters are more dense and well-defined. As you increase the number of clusters (K), the inertia will always decrease because each point will be closer to a centroid.
        - **Davies-Bouldin score:** The average similarity measure of each cluster with its most similar cluster. Similarity is measured as a ratio of within-cluster distances to between-cluster distances. A lower score is better and the minimum score is 0. A lower score indicates that the clusters are more compact and better separated from each other.
        - **Silhouette score:** Provides a more formal way to evaluate the quality of your clustering. It measures how similar an object is to its own cluster (cohesion) compared to other clusters (separation). The score is calculated for each data point and then averaged. The score range is between (-1, 1), where 1 means the data point is well-matched to its own cluster and well-separated from neighboring clusters; 0 means the data point is on or very close to the decision boundary between two clusters; and -1 means the data point is likely assigned to the wrong cluster.
        - **Calinski-Harabasz Index** (Variance Ratio Criterion): This score is a ratio of the between-cluster dispersion mean and the within-cluster dispersion mean. A higher score is better, as it shows that the clusters are dense and well separated.
    - **External Clustering Validation Metrics:** These are used when you have a dataset with known class labels and you want to see how well your clustering algorithm can recreate those classes.
        - **Adjusted Rand Index (ARI):** Measures the similarity between two clusterings (your predicted clusters and the true labels), while the "Adjusted" version corrects for chance. The score ranges from -1 to 1. A score of 1 indicates perfect agreement. A score of 0 indicates a random assignment. A negative score indicates that the assignment was even worse than random assignment.

- **Homogeneity:** Measures if each cluster contains only data points of a single class.
- **Completeness:** Measures if all data points of a given class are in the same cluster.
- **V-measure:** The harmonic mean of homogeneity and completeness, providing a single score that balances both. The score ranges from 0 to 1, with 1 being the best. These metrics are based on conditional entropy.

---

5. Sensitivity

- The model performs a relative classification, meaning it categorizes runners into performance tiers (Beginner / Intermediate / Advanced) based on their results relative to all other runners present in the specific dataset being analyzed.
- Key Implication: A single runner may be classified into different performance tiers when analyzed using different datasets, as the tier assignment is entirely dependent on the competitive distribution of the data.
- Because there is no true predictor, I used the Reigel formula which is calculated from pace and distance, I combined it with average heart rate to create a matrix called GRPS. However, this matrix depends heavily on the calculated pace, which means that classification models may try to "reverse-engineer" the formula instead of learning meaningful patterns.
  In real-world data, we would prefer to classify performance based on ground-truth labels, such as race results or a coach's assessment. Therefore, we will build an experimental model that excludes the calculated pace from its feature set.

In [1]:
```
import os
os.environ['OMP_NUM_THREADS'] = '2'
os.environ["LOKY_MAX_CPU_COUNT"] = "4"
```

In [2]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import plotly.express as px
import datetime
from mpl_toolkits.mplot3d import Axes3D
from sklearn.model_selection import train_test_split, StratifiedKFold, cross_val_predict, learning_curve
from sklearn.inspection import permutation_importance
from sklearn.tree import DecisionTreeClassifier, plot_tree
from sklearn.model_selection import cross_val_score
from sklearn.linear_model import LogisticRegression,LogisticRegressionCV
from sklearn.metrics import roc_auc_score, roc_curve, auc,mean_squared_error,accuracy_score,precision_score,recall_score, f1_s
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay, classification_report
```

```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.cluster import KMeans
from sklearn.preprocessing import QuantileTransformer,PowerTransformer,RobustScaler,StandardScaler,MinMaxScaler
from sklearn.metrics import silhouette_score, davies_bouldin_score, adjusted_rand_score, v_measure_score
from sklearn.ensemble import RandomForestClassifier, GradientBoostingClassifier


print("✅ All libraries are working!")
```

```
✅ All libraries are working!
```

import kagglehub # Download latest version #path = kagglehub.dataset_download("jeffreybraun/running-log-insight") #print("Path to dataset files:", path) path1 = kagglehub.dataset_download("olegoaer/running-races-strava") print("Path to dataset files:", path1) #path2 = kagglehub.dataset_download("ayushtankha/nike-run-club-data-200-runs") #print("Path to dataset files:", path2) ##TCX files, irrelevant #path3 = kagglehub.dataset_download("ajitjadhav1/strava-running-activity-data") #print("Path to dataset files:", path3) ## No HR path4 = kagglehub.dataset_download("purpleyupi/strava-data") print("Path to dataset files:", path4)

In [3]:
```
## Helper Functions
def filter_columns(dfs, columns, names=None):
    new_dfs = []
    for i, df in enumerate(dfs):
        keep = [c for c in columns if c in df.columns]
        if names:
            missing = [c for c in columns if c not in df.columns]
            if missing:
                print(f"{names[i]} is missing columns: {missing}")
        new_dfs.append(df[keep])
    return new_dfs
#Converts pace to float
def pace_to_float(pace):
    if isinstance(pace, str) and ":" in pace:
        minutes, seconds = pace.split(":")
        return int(minutes) + int(seconds)/60
    if isinstance(pace, datetime.time):
        return pace.minute + pace.second / 60
    if isinstance(pace, datetime.timedelta):
        return pace.total_seconds() / 60
    try:
        return float(pace)
    except:
        return None
#Converts time to float
```

```
def time_to_minutes(t):
    if isinstance(t, str) and ":" in t:
        parts = t.split(":")
        try:
            if len(parts) == 3:  # hh:mm:ss
                h, m, s = parts
                return int(h) * 60 + int(m) + float(s) / 60
            elif len(parts) == 2:  # mm:ss
                m, s = parts
                return int(m) + float(s) / 60
            else:
                return float(t)
        except ValueError:
            return None
    if isinstance(t, datetime.timedelta):
        return t.total_seconds() / 60
    if isinstance(t, datetime.time):
        return t.hour*60 + t.minute + t.second / 60
    else:
        return float(t)
#GRPS calculation fuction
def calculate_grps(df):
    required_cols = ["Time", "Distance", "Avg HR"]
    if not all(col in df.columns for col in required_cols):
        raise ValueError(f"Input DataFrame must contain the following columns: {required_cols}")
    df["T2"] = df["Time"] * (10 / df["Distance"]) ** 1.06 # Calculate T2
    df["Normalized_HR"] = df["Avg HR"] / 195 # Calculate Normalized_HR
    # Calculate GRPS
    # To avoid division by zero or large numbers, we add a small epsilon to "T2"
    # This is good practice for numerical stability, especially if "T2" can be zero.
    epsilon = 1e-9
    df["GRPS"] = ((10 / (df["T2"] + epsilon)) / df["Normalized_HR"]) * 100
    return df
```

In [4]:
```
maor = pd.read_csv("data/maor's_data.csv")
ely = pd.read_excel("data/Ely's_data(no_HR).xlsx")
shapira = pd.read_excel("data/shapira's_data.xlsx")
someone = pd.read_csv("data/activity_log.csv")
franco = pd.read_excel("data/franco's_data.xlsx")
danieli = pd.read_excel("data/ido's_data.xlsx")
```

```
ehud = pd.read_excel("data/ehud's_data.xlsx")
#print(maor.head())
columns = ["Distance", "Time", "Avg HR", "Avg Pace"]
dfs = [maor, ely, shapira, someone, franco, danieli, ehud]
names = ["maor", "ely", "shapira", "someone", "franco", "danieli","ehud"]
dfs = filter_columns(dfs, columns, names)
maor, ely, shapira, someone, franco, danieli, ehud = dfs
print("Sample from one of the datasets (shapira's) after filterig the relevant features \n\n", shapira.head())
```

```
Sample from one of the datasets (shapira's) after filterig the relevant features

   Distance      Time Avg HR  Avg Pace
0     19.00  01:50:44    140  05:50:00
1     32.00  02:53:46    150  05:26:00
2     10.01  00:58:27    150  05:50:00
3     19.00  01:35:32    164  05:02:00
4     18.00  01:46:00    141  05:53:00
```

Running Metrics for Classification

I have tried to use a general metric that ignores Vo2 Max (the most common metric for running performance) because I couldn't get that information from all runners. The metric is being calculated for each run by the next formula:

GRPS = (1/Normalized Pace) / Normalized Heart rate (Low pace --> good score. GRPS stands for General running performance score). This metric is very similar to the Riegel model which is used to predict race times for runners by extrapolating from a known race time and distance. The only addition is dividing by the normalized HR so that my formula will consider the runner's average heart rate.

Normalized Pace will be calculated by the 10K Equivalent Pace for each run:

$$T2 = T1(D2/D1)^c$$

- **T1** - Time for Run 1
- **T2** - Predicted time for Run 2
- **D1** - Distance for Run 1
- **D2** - Distance for Run 2 (based on the 'standard' - 10K)
- **c** - A constant, **1.06**. It is based on a well-established model (like the Riegel or Cameron models) that has been validated by analyzing the performance of thousands of runners across different distances.

This effectively accounts for the factor of distance and allows you to compare the "effort" of a 10K race to an easy long run or shorter runs on the same scale.

Normalized Pace = T2/D2

Normalized Heart Rate will be calculated by this formula:

Normalized HR = Avg HR / Benchmark HR

- **Benchmark HR** - 220 - age (Simple but may be inaccurate). For simplification, I used 195 as the benchmark heart rate (all the participants are between the ages of 23-26).

In [5]:
```
for i in range(len(dfs)):
    #df["Avg Pace"] = df["Avg Pace"].apply(pace_to_float)
    df = dfs[i]
    df['Distance'] = df['Distance'].replace(0.0, np.nan)
    df['Avg HR'] = pd.to_numeric(df['Avg HR'], errors='coerce')
    df = df[df["Distance"] < 50].copy() #handles outliers in the data
    df["Time"] = df["Time"].apply(time_to_minutes).astype(float) #Changes the time to a format which is easier to conevrt to f
    df = df.dropna(subset=['Time', 'Distance']) # Drop rows with missing values
    df["Calculated pace"] = df["Time"] / df["Distance"]
    dfs[i] = df
    #print(df.describe())
    #print(df.info())
maor, ely, shapira, someone, franco, danieli, ehud = dfs
```

In [6]:
```
#GRPS calculation
for i in range(len(dfs)):
    dfs[i] = calculate_grps(dfs[i])

maor, ely, shapira, someone, franco, danieli, ehud = dfs
maor_GRPS_score = maor["GRPS"].median()
ely_GRPS_score = ely["GRPS"].median()
someone_GRPS_score = someone["GRPS"].median()
shapira_GRPS_score = shapira["GRPS"].median()
smeone_GRPS_score = someone["GRPS"].median()
franco_GRPS_score = franco["GRPS"].median()
danieli_GRPS_score = danieli["GRPS"].median()
```

```
ehud_GRPS_score = ehud["GRPS"].median()

print("Friends' GRPS scores")
print(f"Maor's GRPS score:, {maor_GRPS_score:.3f}")
print(f"Ely's GRPS score:, {ely_GRPS_score:.3f}")
print(f"Someone's GRPS score:, {someone_GRPS_score:.3f}")
print(f"Shapira's GRPS score:, {shapira_GRPS_score:.3f}")
print(f"Franco's GRPS score:, {franco_GRPS_score:.3f}")
print(f"Danieli's GRPS score:, {danieli_GRPS_score:.3f}")
print(f"Ehud's GRPS score:, {ehud_GRPS_score:.3f}")
```

```
Friends' GRPS scores
Maor's GRPS score:, 23.651
Ely's GRPS score:, 25.735
Someone's GRPS score:, 29.348
Shapira's GRPS score:, 23.790
Franco's GRPS score:, 23.982
Danieli's GRPS score:, 20.792
Ehud's GRPS score:, 21.579
```

In [7]:
```
#Creatimg a median run Data frame with friends' data
median_runs = []
for i, df in enumerate(dfs):
    median_time = df["Time"].median()
    median_distance = df["Distance"].median()
    median_Avg_HR = df["Avg HR"].median()
    median_Calculated_pace = df["Calculated pace"].median()
    median_GRPS = df["GRPS"].median()

    median_df = pd.DataFrame([{
        "Runner": names[i],
        "Median Time": median_time,
        "Median Distance": median_distance,
        "Median Avg HR": median_Avg_HR,
        "Median Calculated Pace": median_Calculated_pace,
        "Median GRPS Score": median_GRPS
    }])
    median_runs.append(median_df)
final_df = pd.concat(median_runs, ignore_index=True)
#print(final_df)
```

Using 2 datasets found on Kaggle. The first one with a total of 116 runners with results of 42000 races, the second one with 165 regular runners.

The two code cells below fit the data format in those datasets to the format of rest of the our data and calculating GRPS to each runner.

In [8]:
```
strava_df = pd.read_excel("datasets\\raw-data-kaggle.xlsx")

# If the data is stored in a single column as semicolon-separated strings, split it
if strava_df.shape[1] == 1 and strava_df.columns[0] == 'athlete;gender;timestamp;distance (m);elapsed time (s);elevation gain
    strava_df = strava_df.iloc[:, 0].str.split(";", expand=True) # Split the single column into multiple columns
    strava_df.columns = ["athlete", "gender", "timestamp", "distance (m)","elapsed time (s)", "elevation gain (m)", "average h

strava_df["timestamp"] = pd.to_datetime(strava_df["timestamp"], dayfirst=True)
strava_df["distance (m)"] = pd.to_numeric(strava_df["distance (m)"], errors='coerce') #If you find a value that can't be conve
strava_df["elapsed time (s)"] = pd.to_numeric(strava_df["elapsed time (s)"], errors='coerce')
strava_df["elevation gain (m)"] = pd.to_numeric(strava_df["elevation gain (m)"], errors='coerce')
strava_df["average heart rate (bpm)"] = pd.to_numeric(strava_df["average heart rate (bpm)"], errors='coerce')
strava_df["Distance"] = strava_df["distance (m)"] / 1000
strava_df["Time"] = strava_df["elapsed time (s)"] / 60
strava_df["Calculated Pace"] = strava_df["Time"] / strava_df["Distance"]
strava_df.drop(columns = ["gender","elevation gain (m)","distance (m)", "timestamp", "elapsed time (s)"], inplace = True)
strava_df.rename(columns={"average heart rate (bpm)": "Avg HR"}, inplace=True)
strava_df.rename(columns={"athlete": "Runner"}, inplace=True)

#print(strava_df.head())
median_strava_runs = strava_df.groupby("Runner", as_index=False)[["Avg HR", "Distance", "Time", "Calculated Pace"]].median()
median_strava_runs["Avg HR"] = strava_df.groupby("Runner")["Avg HR"].transform(lambda x: x.fillna(x.median()))
median_strava_runs = calculate_grps(median_strava_runs)
median_strava_runs.drop(columns = ["Normalized_HR","T2"], inplace = True)
#print(median_strava_runs.sample(7))
#print(median_strava_runs.info())
#print(median_strava_runs.describe())
```

In [9]:
```
strava2_df = pd.read_csv("datasets\\strava_full_data.csv")
strava2_df.drop(columns = ["kudos_count","start_date_local","type","elev_high", "max_heartrate", "max_speed","elapsed_time","t
#strava2_df["distance"] = pd.to_numeric(strava2_df["distance"], errors='coerce') #If you find a value that can't be converted,
strava2_df["moving_time"] = strava2_df["moving_time"].apply(time_to_minutes).astype(float)
strava2_df["average_heartrate"] = pd.to_numeric(strava2_df["average_heartrate"], errors='coerce')
strava2_df["distance"] = strava2_df["distance"] / 1000
strava2_df.rename(columns={"distance": "Distance"}, inplace=True)
```

```
strava2_df.rename(columns={"moving_time": "Time"}, inplace=True)
strava2_df["Calculated Pace"] = strava2_df["Time"] / strava2_df["Distance"]
strava2_df.rename(columns={"average_heartrate": "Avg HR"}, inplace=True)
strava2_df.rename(columns={"Column1": "Runner"}, inplace=True)
#print(strava_df.head())

median_strava2_runs = strava2_df.groupby("Runner", as_index=False)[["Avg HR", "Distance", "Time", "Calculated Pace"]].median()
median_strava2_runs = median_strava2_runs.dropna(subset=["Avg HR"])
median_strava2_runs = calculate_grps(median_strava2_runs)
median_strava2_runs.drop(columns = ["Normalized_HR","T2"], inplace = True)
median_strava2_runs = median_strava2_runs.loc[median_strava2_runs['GRPS'] < 35] #handles outliers in the data
median_strava2_runs = median_strava2_runs.loc[median_strava2_runs['Calculated Pace'] < 9] #handles outliers in the data

#print(median_strava2_runs.sample(7))
#print(median_strava2_runs.info())
#print(median_strava2_runs.describe())
```

Fitting the format of both strava data frames to the same format as the origianl data frame. Adding labels to thr final data frame.

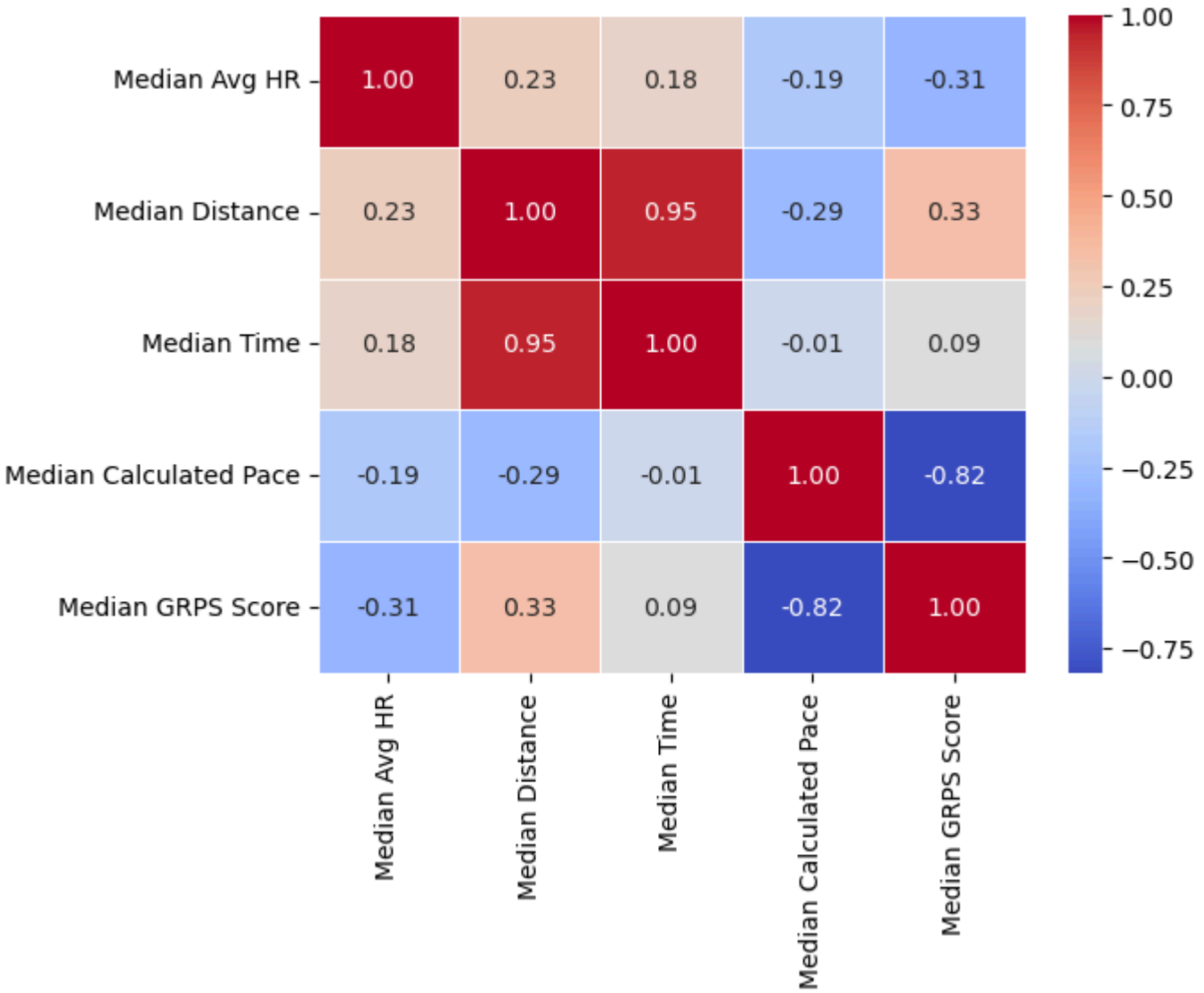
In [10]:
```
final_strava = pd.concat([median_strava_runs, median_strava2_runs], ignore_index = True)
final_strava.rename(columns={"Avg HR": "Median Avg HR"}, inplace=True)
final_strava.rename(columns={"Distance": "Median Distance"}, inplace=True)
final_strava.rename(columns={"Time": "Median Time"}, inplace=True)
final_strava.rename(columns={"Calculated Pace": "Median Calculated Pace"}, inplace=True)
final_strava.rename(columns={"GRPS": "Median GRPS Score"}, inplace=True)
final_df = pd.concat([final_strava, final_df], ignore_index = True)
final_df["Runner_Class"] = pd.qcut(final_df["Median GRPS Score"], q=[0, 0.30, 0.80, 1], labels=["Beginner", "Intermediate", "A
#print(final_df.tail(7))
#print(final_df.info())
```

In [11]:
```
final_df = final_df.loc[final_df['Median Time'] < 150]
final_df = final_df.loc[final_df['Median Calculated Pace'] < 9.5]
print("Information of our finalize data frame \n")
print(final_df.info())
```

```
Information of our finalize data frame

<class 'pandas.core.frame.DataFrame'>
Index: 286 entries, 0 to 287
Data columns (total 7 columns):
 #   Column                 Non-Null Count  Dtype
---  ------                 --------------  -----
 0   Runner                 286 non-null    object
 1   Median Avg HR          286 non-null    float64
 2   Median Distance        286 non-null    float64
 3   Median Time            286 non-null    float64
 4   Median Calculated Pace  286 non-null    float64
 5   Median GRPS Score      286 non-null    float64
 6   Runner_Class           286 non-null    category
dtypes: category(1), float64(5), object(1)
memory usage: 16.0+ KB
None
```

In [12]:
```
numeric_cols = final_df.select_dtypes(include=np.number)
corr_mat = numeric_cols.corr()
print("Correlation matrix between the data frame features")
print(sns.heatmap(corr_mat, annot=True, cmap="coolwarm", fmt=".2f", linewidths=0.5))
```

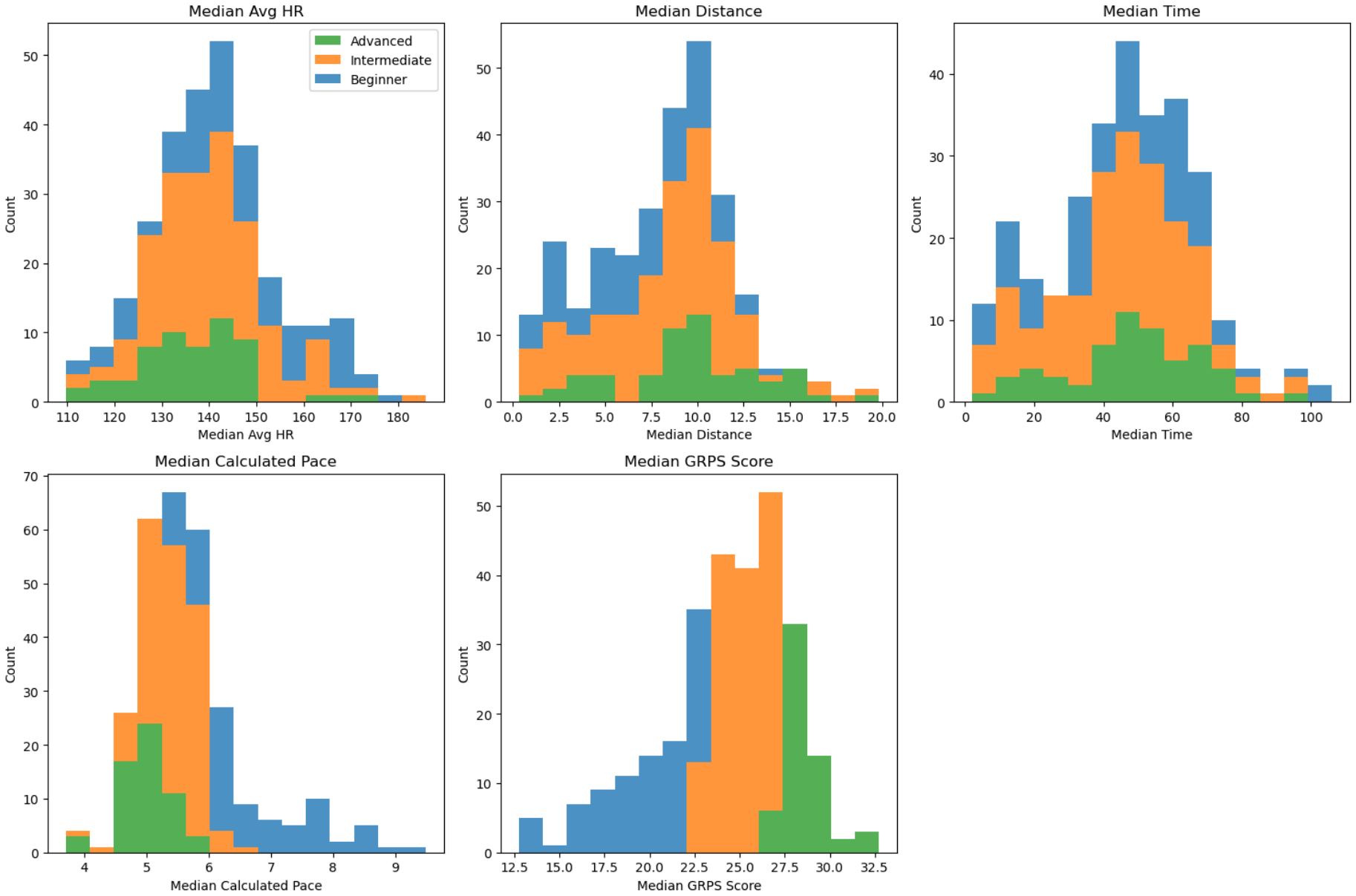
```
Correlation matrix between the data frame features
Axes(0.125,0.11;0.62x0.77)
```



In [13]:
```
runner_classes = ["Advanced", "Intermediate", "Beginner"]
colors = ['tab:green', 'tab:orange', 'tab:blue']
labels = ['Advanced', 'Intermediate', 'Beginner']
numeric_cols = final_df.select_dtypes(include=np.number).columns.tolist()
num_plots = len(numeric_cols)
num_cols = 3
```

```
num_rows = (num_plots + num_cols - 1) // num_cols
fig, axes = plt.subplots(num_rows, num_cols, figsize=(15, 10))
axes = axes.flatten()
print("Feature histogrms:\n")
for i, col in enumerate(numeric_cols):
    ax = axes[i]
    data = [final_df[final_df['Runner_Class'] == cls][col] for cls in runner_classes]
    ax.hist(data, bins=15, stacked=True, color=colors, label=labels, alpha=0.8)
    ax.set_title(col)
    ax.set_xlabel(col)
    ax.set_ylabel('Count')
    if i == 0:
        ax.legend()
for j in range(i + 1, len(axes)):
    fig.delaxes(axes[j])
plt.tight_layout()
plt.show()
```

```
Feature histogrms:
```



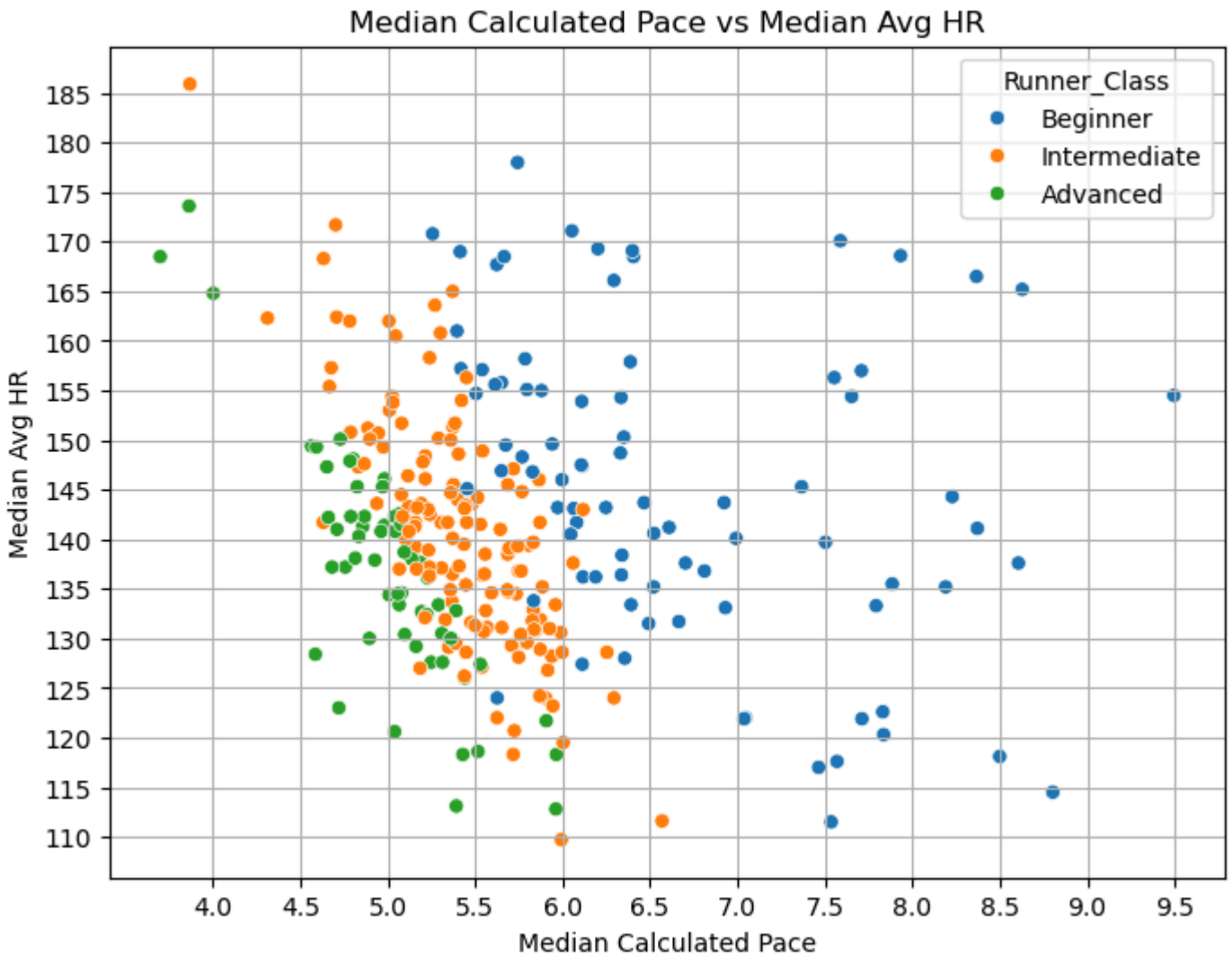
In [14]:
```
final_df_model = final_df.drop(columns = ["Median GRPS Score"])
train_split, test_split = train_test_split(final_df_model, test_size = 0.2, random_state = 42, shuffle = True)
df_train_features = train_split
df_test_features = test_split
```

```
df_train_target = df_train_features["Runner_Class"]
df_test_target = df_test_features["Runner_Class"]
#print(df_train_features.info())
#print(df_test_features)
#print(df_train_target)
#print(df_test_target)
```

Exploring the features correlations: Distance, Calculated pace, avg HR in crossbreeding with Runner Class

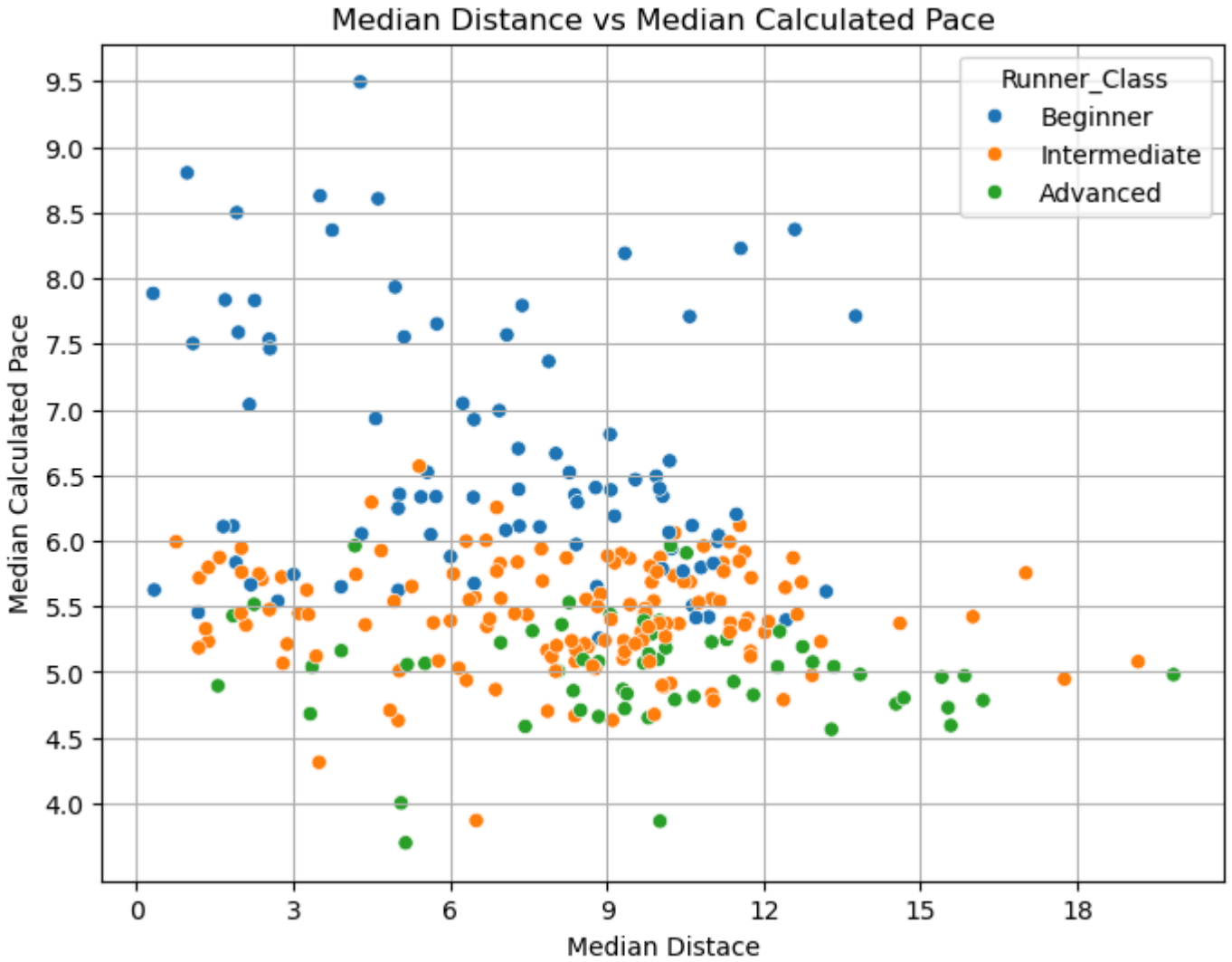
I didn't use Median Time because it and Median Distance are extremly high correlated

In [15]:
```
#Calculated Pace and Avg HR in correlation with Runner Class
plt.figure(figsize=(8,6))
sns.scatterplot(data=final_df_model, x="Median Calculated Pace", y="Median Avg HR", hue = "Runner_Class" )
plt.title("Median Calculated Pace vs Median Avg HR")
plt.xlabel("Median Calculated Pace")
plt.ylabel("Median Avg HR")
tick_locations = np.arange(4, 10, 0.5)
tick_y_locations = np.arange(110,190,5)
plt.xticks(tick_locations)
plt.yticks(tick_y_locations)
plt.grid(True)
plt.show()
```



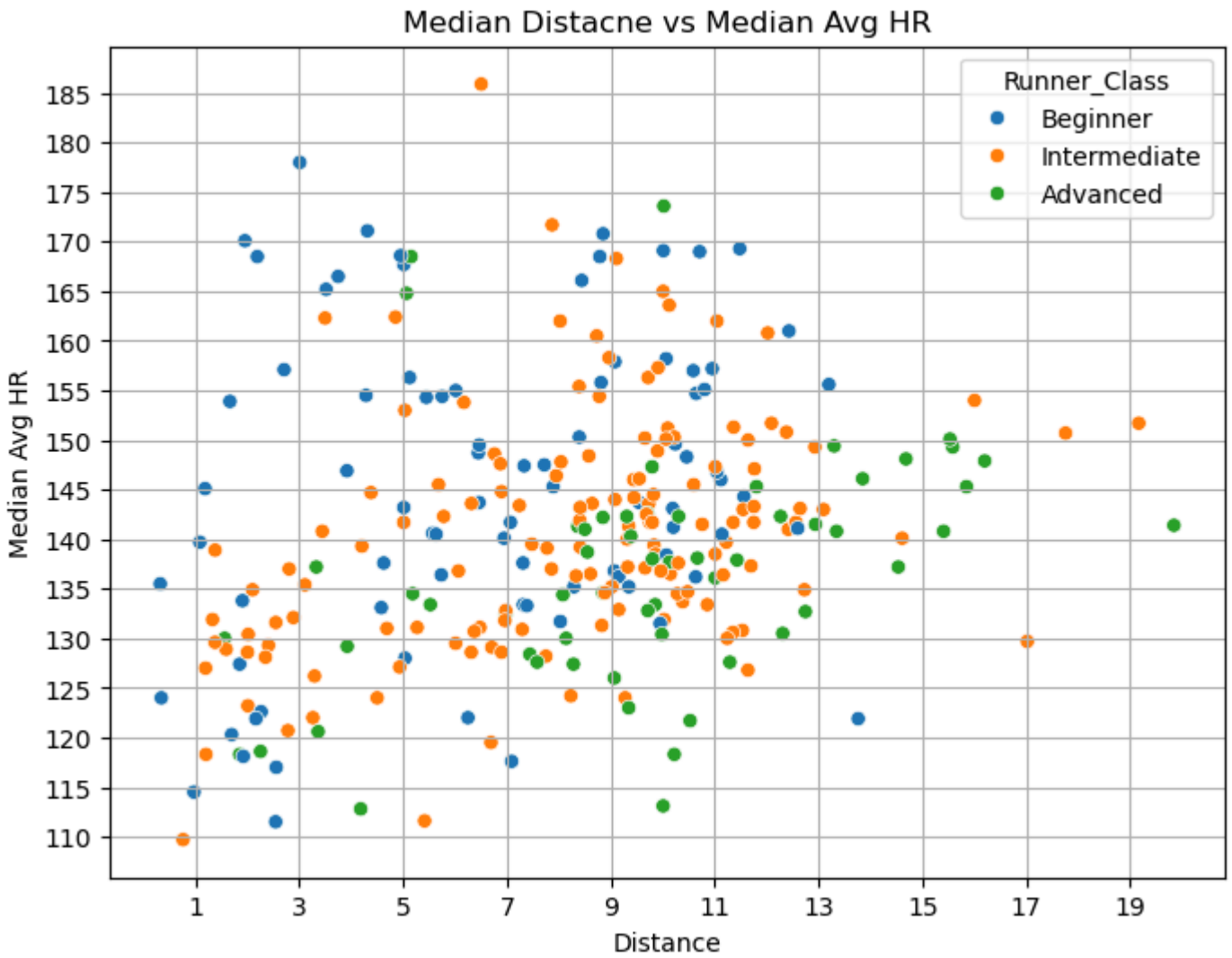
In [16]:
```
#Distance and Pace in correlation with Runner Class
plt.figure(figsize=(8,6))
sns.scatterplot(data=final_df_model, x="Median Distance", y="Median Calculated Pace", hue = "Runner_Class" )
plt.title("Median Distance vs Median Calculated Pace")
plt.xlabel("Median Distace")
plt.ylabel("Median Calculated Pace")
tick_locations = np.arange(0, 21, 3)
```

```
tick_y_locations = np.arange(4,10,0.5)
plt.xticks(tick_locations)
plt.yticks(tick_y_locations)
plt.grid(True)
plt.show()
```



In [17]:
```
#Avg HR and Distacne in correlation with Runner Class
plt.figure(figsize=(8,6))
```

```
sns.scatterplot(data=final_df_model, x="Median Distance", y="Median Avg HR", hue = "Runner_Class" )
plt.title("Median Distacne vs Median Avg HR")
plt.xlabel("Distance")
plt.ylabel("Median Avg HR")
tick_locations = np.arange(1, 21, 2)
tick_y_locations = np.arange(110,190,5)
plt.xticks(tick_locations)
plt.yticks(tick_y_locations)
plt.grid(True)
plt.show()
```



Data exploring conclusions

- We can see a very high correlation (0.95) between "Time" and "Distance." This strong relationship suggests we may only need to use one of these features in our models to avoid multicollinearity.
- The absolute correlation between "GRPS" and "Calculated Pace" is the highest among all other features relative to "GRPS."

- From the graphs above, it appears that "Calculated Pace" is the feature that most clearly separates the different classes. Other features like "Average HR," "Distance," and "Time" are more widely spread, with similar values across all classes. However, it's worth noting that specific regions of combined data (for example, "Calculated Pace" lower than 5.1 and "Distance" larger than 13) can still effectively characterize a class.

1st classification method - Manual Desicion Tree

Based on the data, "Calculated Pace" appears to be the feature that most clearly separates the different classes. This led us to assign it the highest importance in our manually constructed decision tree. Subsequently, I attempted to further distinguish the classes using "Average HR" and "Distance."

In [18]:
```
def decision_tree_rules(runner_row):
    if runner_row["Median Calculated Pace"] >= 6.05:
        return "Beginner"
    elif runner_row["Median Calculated Pace"] <= 5.1:
        if runner_row["Median Distance"] >= 13:
            return "Advanced"
        elif runner_row["Median Avg HR"]<= 152:
            return "Advanced"
        else:
            return "Intermediate"
    elif runner_row["Median Calculated Pace"] <=6.05:
        if 120 <= runner_row["Median Avg HR"] <= 165:
            return "Intermediate"
        elif runner_row["Median Avg HR"] <= 120:
            return "Advanced"
    return "Beginner"

res_test = df_test_features.apply(decision_tree_rules, axis=1)
res_train = df_train_features.apply(decision_tree_rules, axis=1)
```

In [19]:
```
#Accuracy test split:
result_test = df_test_target == res_test
count_T_and_F = result_test.value_counts()
accuracy = count_T_and_F[True] / len(res_test)
print("Test Accuracy:", accuracy)
print(count_T_and_F)
```

```
#Accuracy train split:
result_train = df_train_target == res_train
count_T_and_F = result_train.value_counts()
accuracy = count_T_and_F[True] / len(res_train)
print("Train Accuracy:", accuracy)
print(count_T_and_F)
```

```
Test Accuracy: 0.7413793103448276
True     43
False    15
Name: count, dtype: int64
Train Accuracy: 0.793859649122807
True     181
False     47
Name: count, dtype: int64
```

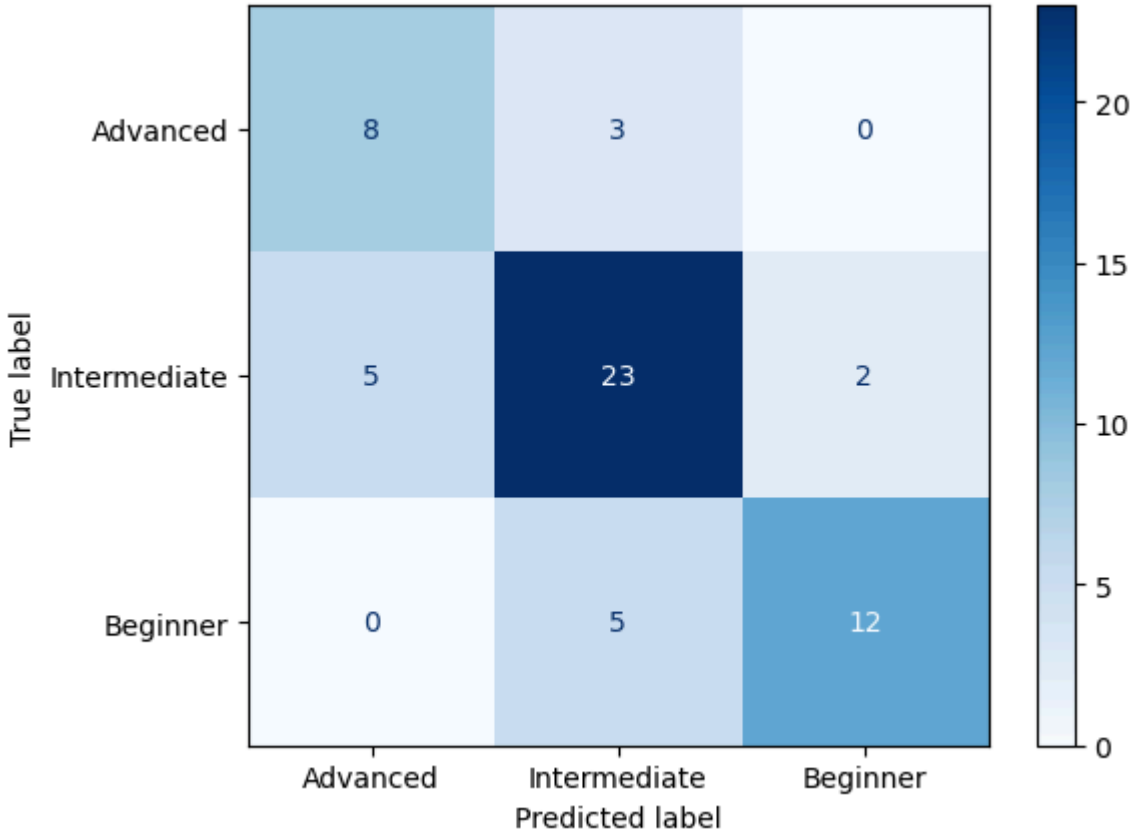
1st Method Results

In [20]:
```
print("Classification report - manual Decision Tree:")
print(classification_report(df_test_target, res_test))
```

```
Classification report - manual Decision Tree:
              precision    recall  f1-score   support

    Advanced       0.62      0.73      0.67        11
    Beginner       0.86      0.71      0.77        17
Intermediate       0.74      0.77      0.75        30

    accuracy                           0.74        58
   macro avg       0.74      0.73      0.73        58
weighted avg       0.75      0.74      0.74        58
```

In [21]:
```
print("Confusion matrix - manual Decision Tree:")
labels_cm = ['Advanced', 'Intermediate', 'Beginner']
cm_DT_manual = confusion_matrix(df_test_target, res_test, labels=labels_cm)
disp = ConfusionMatrixDisplay(confusion_matrix=cm_DT_manual, display_labels=labels_cm)
disp.plot(cmap=plt.cm.Blues)
plt.show()
```

```
Confusion matrix - manual Decision Tree:
```



2nd Classification Method - SKL Decision Tree

Using the decision tree built in function of SKL

In [22]:
```
df_train_features_numeric = df_train_features.select_dtypes(include=[int, float])
df_test_features_numeric = df_test_features.select_dtypes(include=[int, float])
sk_tree = DecisionTreeClassifier(random_state=42, max_leaf_nodes=5)
sk_tree.fit(df_train_features_numeric, df_train_target)

# Predict
y_predict_train_sklearn = sk_tree.predict(df_train_features_numeric)
y_predict_test_sklearn = sk_tree.predict(df_test_features_numeric)

accuracy_train_sklearn = accuracy_score(df_train_target, y_predict_train_sklearn)
```

```
accuracy_test_sklearn = accuracy_score(df_test_target, y_predict_test_sklearn)

print("Sklearn Decision Tree Accuracy on train set:", accuracy_train_sklearn)
print("Sklearn Decision Tree Accuracy on test set:", accuracy_test_sklearn)
```

```
Sklearn Decision Tree Accuracy on train set: 0.8289473684210527
Sklearn Decision Tree Accuracy on test set: 0.603448275862069
```

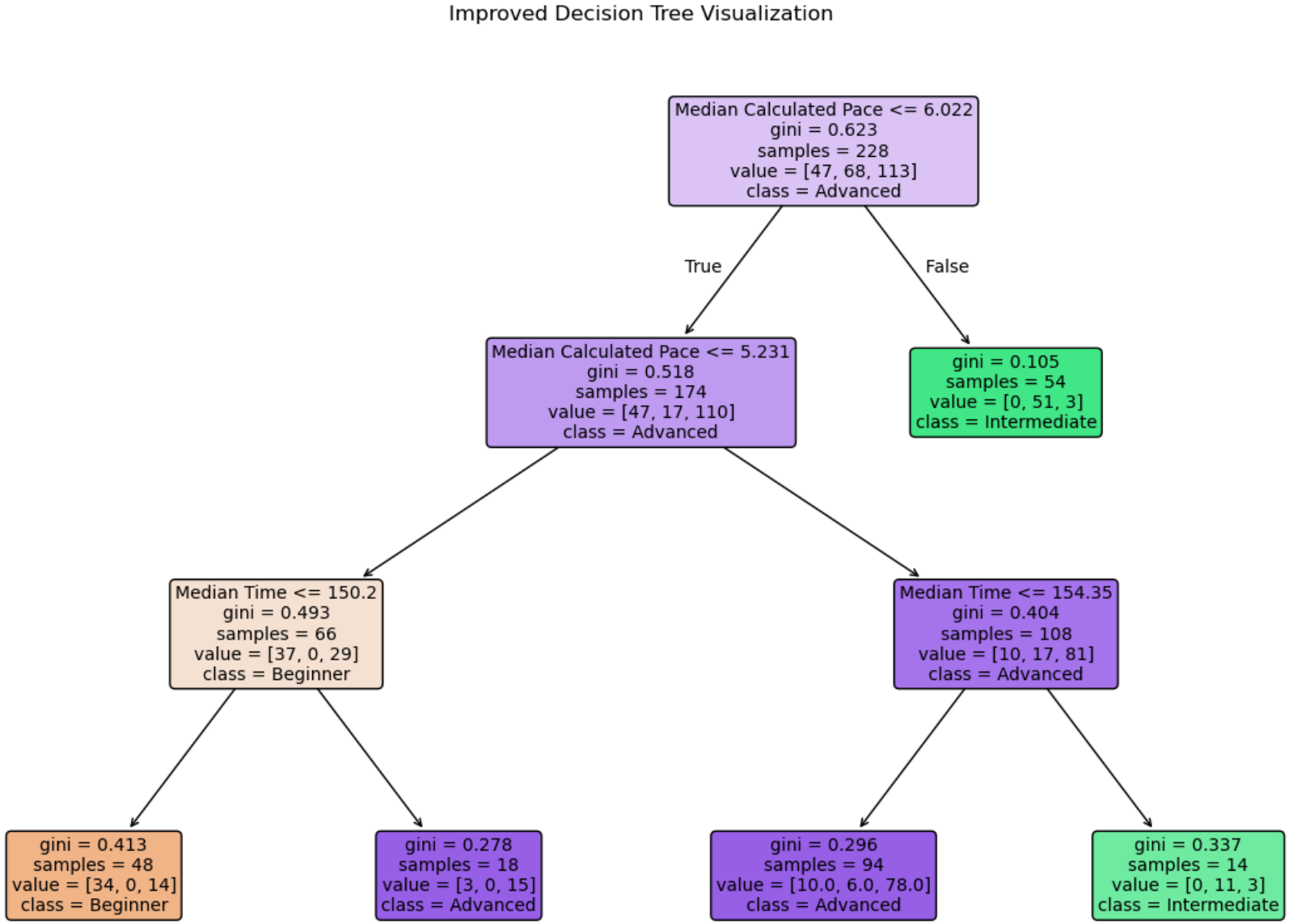
2nd Method Results

In [23]:
```
print("Classification report - SKL Decision Tree:")
print(classification_report(df_test_target, y_predict_test_sklearn))
#print(classification_report(df_train_target, y_predict_train_sklearn))
```

```
Classification report - SKL Decision Tree:
              precision    recall  f1-score   support

    Advanced       0.39      0.64      0.48        11
    Beginner       0.76      0.76      0.76        17
Intermediate       0.65      0.50      0.57        30

    accuracy                           0.60        58
   macro avg       0.60      0.63      0.60        58
weighted avg       0.64      0.60      0.61        58
```

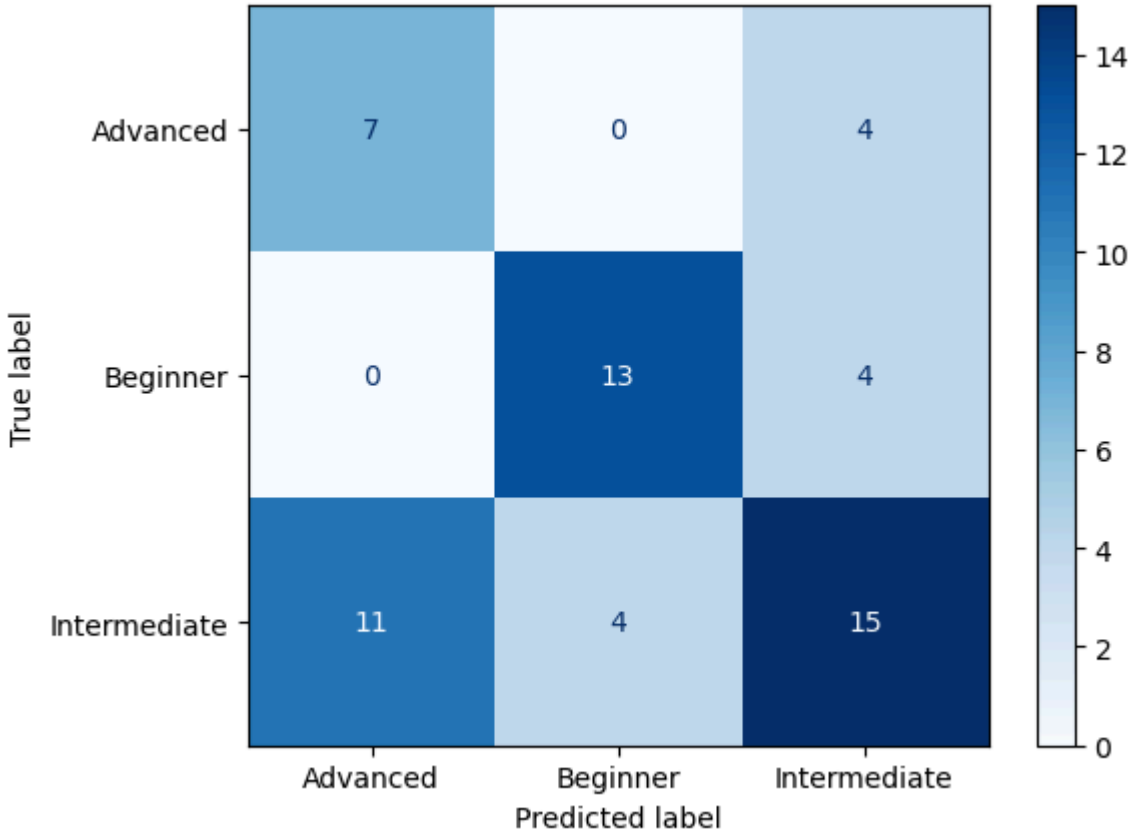
In [24]:
```
feature_names = ["Median Time", "Median Distance", "Median Avg HR", "Median Calculated Pace", "Median GRPS score"]
class_names = ["Beginner", "Intermediate", "Advanced"]

plt.figure(figsize=(15, 10))
plot_tree(sk_tree,
          feature_names=feature_names,
          class_names=class_names,
          filled=True,
          rounded=True,
          fontsize=10,
          impurity=True,
          node_ids=False,
          proportion=False,
          precision=3)
```

```
plt.title("Improved Decision Tree Visualization")
plt.show()
```



In [25]:
```
print("Confusion matrix - SKL Decision Tree:")
cm_DT_func = confusion_matrix(df_test_target, y_predict_test_sklearn, labels=sk_tree.classes_)
disp = ConfusionMatrixDisplay(confusion_matrix=cm_DT_func, display_labels=sk_tree.classes_)
disp.plot(cmap=plt.cm.Blues)
plt.show()
```

```
Confusion matrix - SKL Decision Tree:
```



3rd Method - Logistic Regression

Using the built in function of SKL and using Cross Validiation to evaluate the performance of the model on unseen sets of data. We should use cross validation to avoid over-fitting and under-fitting, and to get a more reliable estimate of model perfomance.

In [26]:
```
#df_train_features_numeric_LR = df_train_features_numeric.drop(columns = ["Median Time"])
#df_test_features_numeric_LR = df_test_features_numeric.drop(columns = ["Median Time"])
```

```
pipe = Pipeline([('scaler', StandardScaler()),('logreg', LogisticRegressionCV(max_iter=300, random_state=42))])
pipe.fit(df_train_features_numeric, df_train_target)
y_pred = pipe.predict(df_test_features_numeric)
coefficients = pd.DataFrame({'Feature': df_train_features_numeric.columns, 'Coefficient': pipe[1].coef_[0]})
print("Model's coefficients:")
print(coefficients)
print("Intercept:", pipe[1].intercept_)
```

```
Model's coefficients:
                  Feature  Coefficient
0           Median Avg HR    -9.588079
1         Median Distance    14.269196
2             Median Time   -11.191895
3  Median Calculated Pace   -14.740097
Intercept: [-10.18918617   3.59518833   6.59399783]
```

Scaling the data is crucial in Logistic Regression mainly because we want to avoid from the impact of mismatched scale

3rd Method Results

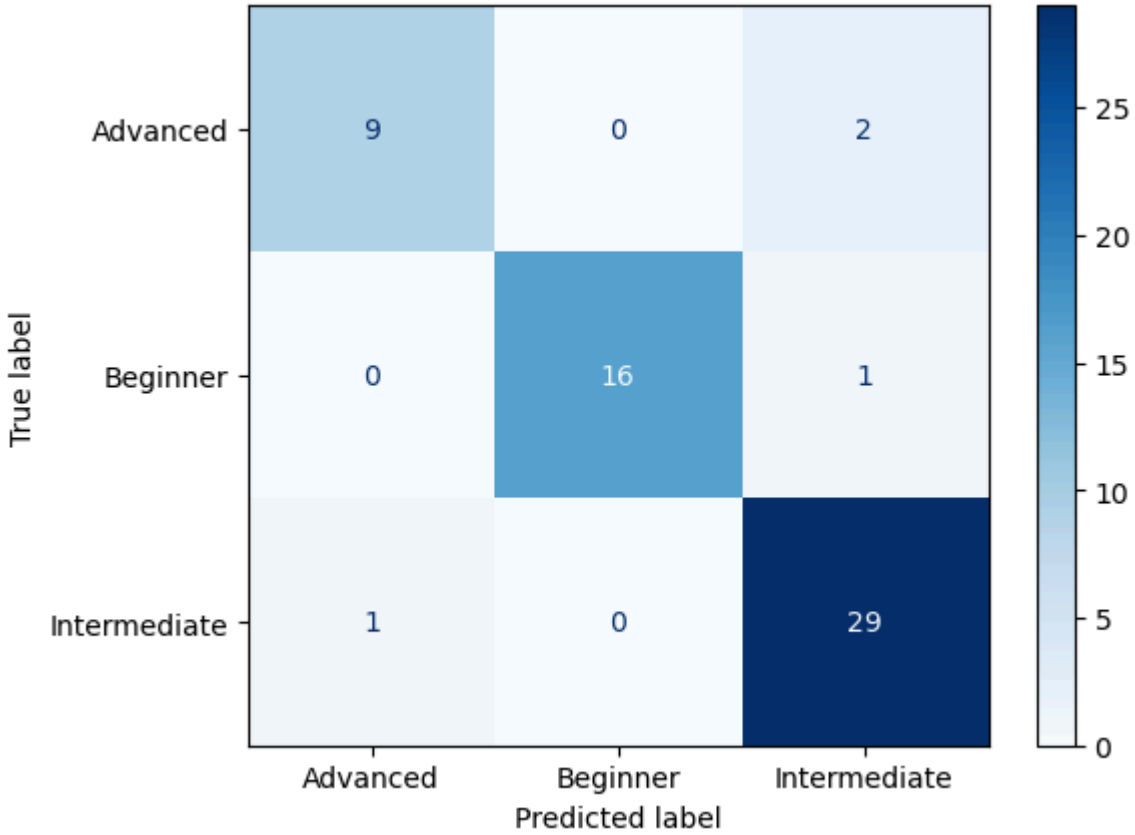
In [27]:
```
print("Classification report - Logistic Regression:")
print(classification_report(df_test_target, y_pred))
train_pred = pipe.predict(df_train_features_numeric)
test_pred = pipe.predict(df_test_features_numeric)
train_pred_proba = pipe.predict_proba(df_train_features_numeric)
test_pred_proba = pipe.predict_proba(df_test_features_numeric)

train_accuracy = accuracy_score(df_train_target, train_pred)
test_accuracy = accuracy_score(df_test_target, test_pred) ## accuracy - total of True results out of all results (TP + TN / TP

train_roc_auc = roc_auc_score(df_train_target, train_pred_proba, multi_class='ovr')
test_roc_auc = roc_auc_score(df_test_target, test_pred_proba, multi_class='ovr') #roc_AUC - area under the roc curve

train_precision = precision_score(df_train_target, train_pred, average='weighted')
test_precision = precision_score(df_test_target, test_pred, average='weighted') #Precision - How much of the positive is reall

train_recall = recall_score(df_train_target, train_pred, average='weighted')
test_recall = recall_score(df_test_target, test_pred, average='weighted') #Recall - how much of the positive we have predicted

print("Train Accuracy:", train_accuracy)
print("Test Accuracy:", test_accuracy)
```

```
print("Train ROC AUC:", train_roc_auc)
print("Test ROC AUC:", test_roc_auc)
print("Train Precision:", train_precision)
print("Test Precision:", test_precision)
print("Train Recall:", train_recall)
print("Test Recall:", test_recall)
```

```
Classification report - Logistic Regression:
              precision    recall  f1-score   support

    Advanced       0.90      0.82      0.86        11
    Beginner       1.00      0.94      0.97        17
Intermediate       0.91      0.97      0.94        30

    accuracy                           0.93        58
   macro avg       0.94      0.91      0.92        58
weighted avg       0.93      0.93      0.93        58

Train Accuracy: 0.9166666666666666
Test Accuracy: 0.9310344827586207
Train ROC AUC: 0.9880578379060628
Test ROC AUC: 0.981697675846612
Train Precision: 0.9167710944026733
Test Precision: 0.9325431034482758
Train Recall: 0.9166666666666666
Test Recall: 0.9310344827586207
```

In [28]:
```
print("Confusion matrix - Logistic Regression:")
cm_LR = confusion_matrix(df_test_target, test_pred, labels=pipe.classes_)
disp = ConfusionMatrixDisplay(confusion_matrix=cm_LR, display_labels=pipe.classes_)
disp.plot(cmap=plt.cm.Blues)
plt.show()
```

```
Confusion matrix - Logistic Regression:
```



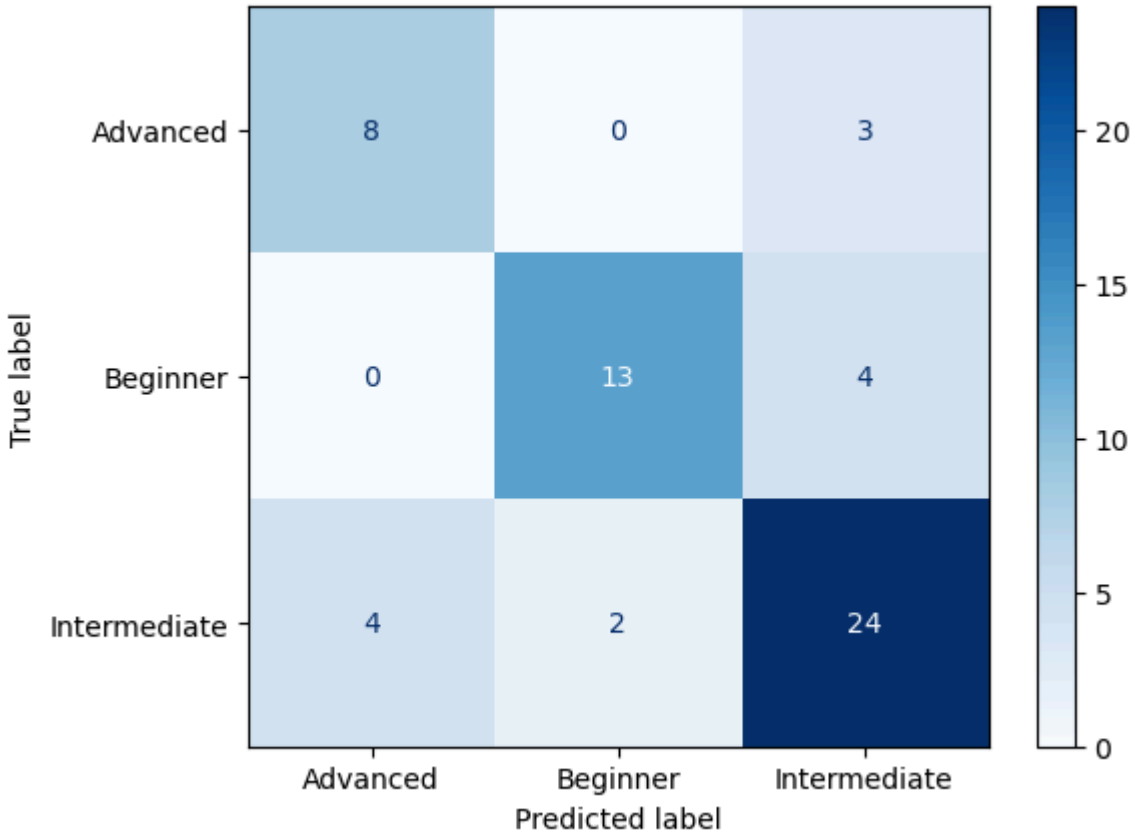
4th Classification Method - Random Forest - An ensemble of decision trees. Each tree is trained on a bootstrap sample (random subset with replacement). At each split, only a random subset of features is considered.Final prediction = majority vote (classification) or average (regression). This randomness reduces variance (overfitting), while maintaining low bias.
Advantages - Handles nonlinear feature interactions well. Robust to outliers & noise. Provides feature importance. **Cross-val predictions** on training data (for diagnostics) - This process ensures that every data point gets predicted once, but never by a model trained on itself.

In [29]:
```
# Random forest Pipeline
features = final_df_model.select_dtypes(include=[int, float])
feature_cols = features.columns.tolist()
rf_pipe = Pipeline([
    ("scaler", StandardScaler()),
    ("rf", RandomForestClassifier(
```

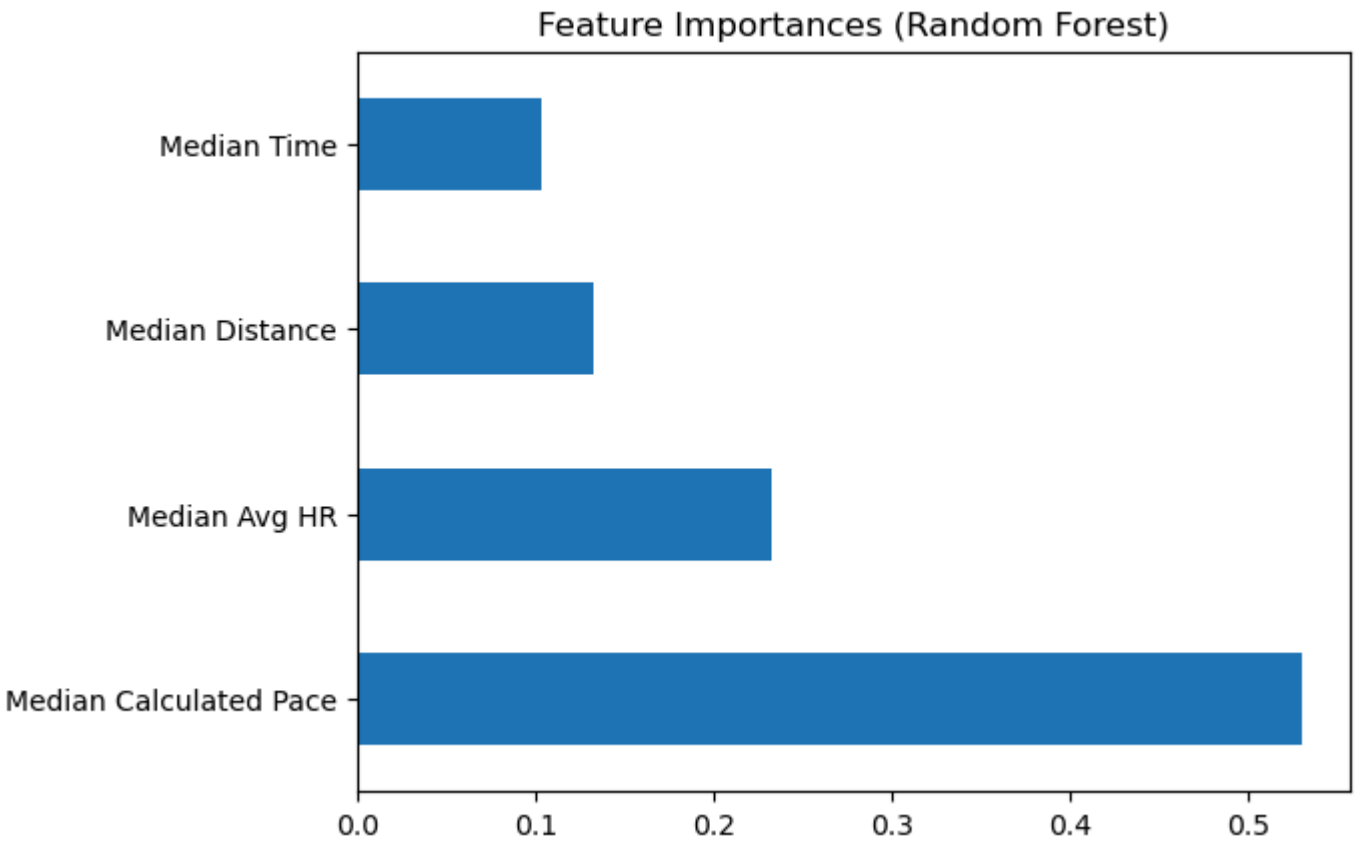
```
        n_estimators=200, max_depth=None, random_state=42
    ))
])

cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
y_pred_cv = cross_val_predict(rf_pipe, df_train_features_numeric, df_train_target, cv=cv)
print("CV classification report - Random Forest:")
print(classification_report(df_train_target, y_pred_cv, digits=3))

rf_pipe.fit(df_train_features_numeric, df_train_target)
y_pred_test = rf_pipe.predict(df_test_features_numeric)
print("Test results classification report - Random Forest:")
print(classification_report(df_test_target, y_pred_test, digits=3))

train_accuracy = accuracy_score(df_train_target, y_pred_cv)
test_accuracy = accuracy_score(df_test_target, y_pred_test)
```

```
CV classification report - Random Forest:
              precision    recall  f1-score   support

    Advanced      0.794     0.574     0.667        47
    Beginner      0.875     0.824     0.848        68
Intermediate      0.754     0.867     0.807       113

    accuracy                          0.794       228
   macro avg      0.808     0.755     0.774       228
weighted avg      0.798     0.794     0.790       228

Test results classification report - Random Forest:
              precision    recall  f1-score   support

    Advanced      0.667     0.727     0.696        11
    Beginner      0.867     0.765     0.812        17
Intermediate      0.774     0.800     0.787        30

    accuracy                          0.776        58
   macro avg      0.769     0.764     0.765        58
weighted avg      0.781     0.776     0.777        58
```

In [30]:
```
cm = confusion_matrix(df_test_target, y_pred_test, labels=rf_pipe.classes_)
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=rf_pipe.classes_)
disp.plot(cmap="Blues")
plt.show()
```



In [31]:
```
importances = rf_pipe.named_steps["rf"].feature_importances_
feat_imp = pd.Series(importances, index=feature_cols).sort_values(ascending=False)
feat_imp.plot(kind="barh")
plt.title("Feature Importances (Random Forest)")
plt.show()
```



Fifth Method - Gradient Boosting Another ensemble of trees, but instead of training independently (like RF), each new tree is trained to correct the errors of the previous trees.It's sequential → learns residuals. The model improves gradually, with a learning rate controlling how much each new tree contributes.

Advantages - Typically achieves higher accuracy than Random Forest if tuned well. Handles complex nonlinear interactions. More sensitive to hyperparameters (learning_rate, n_estimators, max_depth). Often best for tabular data

In [32]:
```
#Gradient boosting
gb_pipe = Pipeline([
    ("scaler", StandardScaler()),
    ("gb", GradientBoostingClassifier(
        n_estimators=300, learning_rate=0.05, max_depth=3, random_state=42
    ))
```

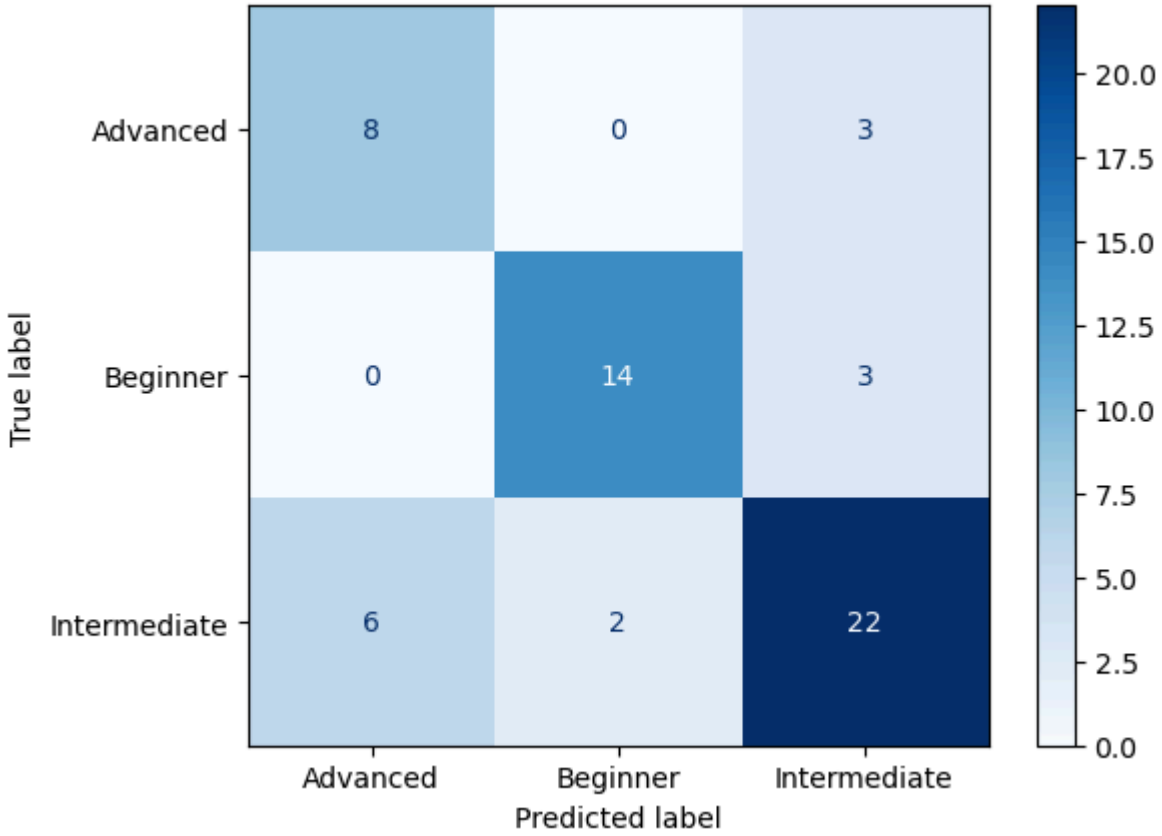
```
])

y_pred_cv = cross_val_predict(gb_pipe, df_train_features_numeric, df_train_target, cv=cv)
print("CV results - Classification report - Gradient Boosting:")
print(classification_report(df_train_target, y_pred_cv, digits=3))

gb_pipe.fit(df_train_features_numeric, df_train_target)
y_pred_test = gb_pipe.predict(df_test_features_numeric)
print("Test results - Classification report - Gradient Boosting:")
print(classification_report(df_test_target, y_pred_test, digits=3))

train_accuracy = accuracy_score(df_train_target, y_pred_cv)
test_accuracy = accuracy_score(df_test_target, y_pred_test)
```

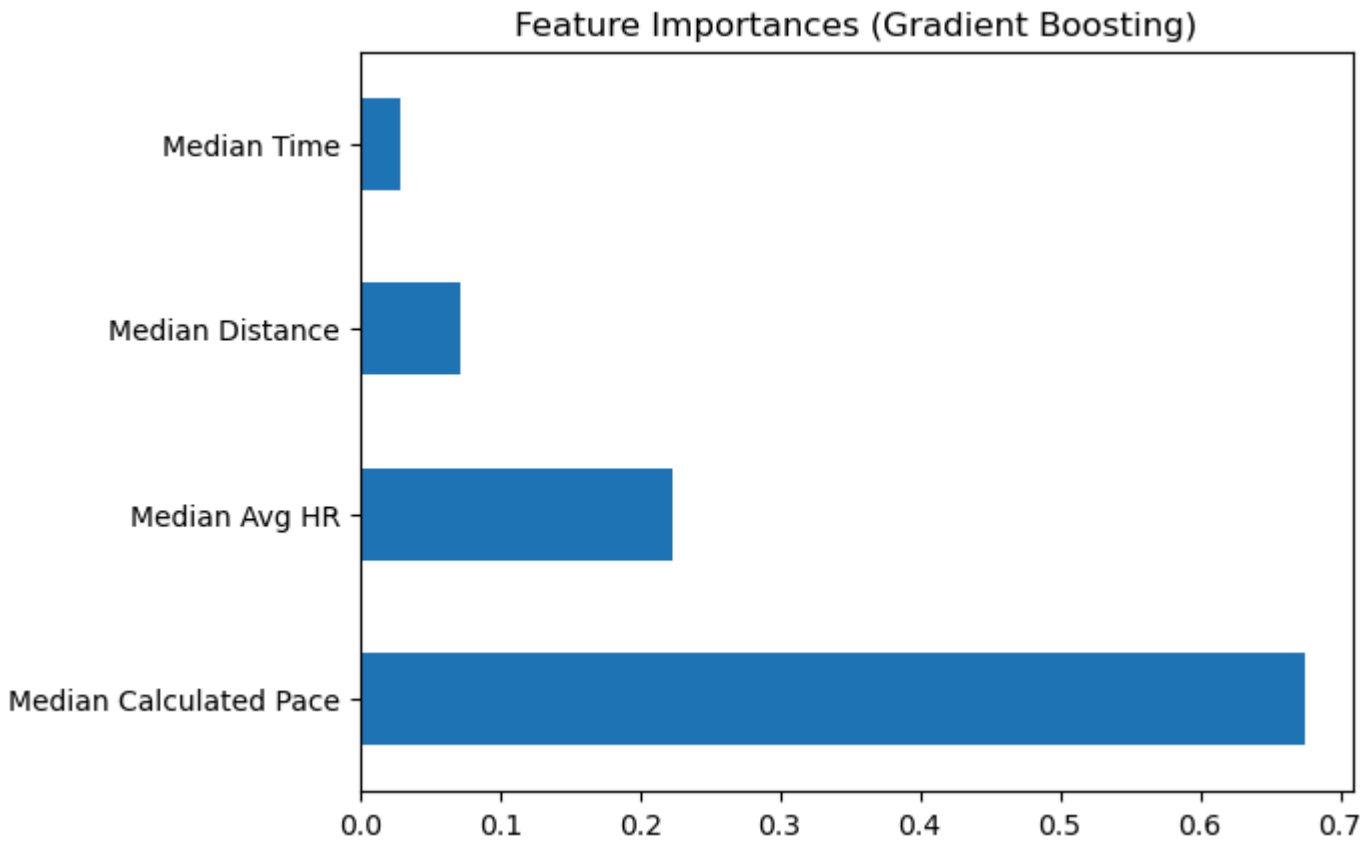
```
CV results - Classification report - Gradient Boosting:
              precision    recall  f1-score   support

    Advanced      0.756     0.660     0.705        47
    Beginner      0.859     0.809     0.833        68
Intermediate      0.764     0.832     0.797       113

    accuracy                          0.789       228
   macro avg      0.793     0.767     0.778       228
weighted avg      0.791     0.789     0.789       228

Test results - Classification report - Gradient Boosting:
              precision    recall  f1-score   support

    Advanced      0.571     0.727     0.640        11
    Beginner      0.875     0.824     0.848        17
Intermediate      0.786     0.733     0.759        30

    accuracy                          0.759        58
   macro avg      0.744     0.761     0.749        58
weighted avg      0.771     0.759     0.762        58
```

In [33]:
```
print("Confusion matrix - Gradient Boosting")
cm = confusion_matrix(df_test_target, y_pred_test, labels=gb_pipe.classes_)
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=gb_pipe.classes_)
disp.plot(cmap="Blues")
```

```
plt.show()

importances = gb_pipe.named_steps["gb"].feature_importances_
feat_imp = pd.Series(importances, index=feature_cols).sort_values(ascending=False)
feat_imp.plot(kind="barh")
plt.title("Feature Importances (Gradient Boosting)")
plt.show()
```

```
Confusion matrix - Gradient Boosting
```





In [34]:
```
def eval_model(pipe, X_train, X_test, y_train, y_test, model_name):
    pipe.fit(X_train, y_train)
    y_pred = pipe.predict(X_test)
    return {
        "Model": model_name,
        "Accuracy": accuracy_score(y_test, y_pred),
        "Macro F1": f1_score(y_test, y_pred, average="macro"),
        "Macro Precision": precision_score(y_test, y_pred, average="macro"),
        "Macro Recall": recall_score(y_test, y_pred, average="macro"),
    }

results = []
results.append(eval_model(sk_tree, df_train_features_numeric, df_test_features_numeric, df_train_target, df_test_target, "SKL
results.append(eval_model(pipe, df_train_features_numeric, df_test_features_numeric, df_train_target, df_test_target, "Logisti
```

```
results.append(eval_model(rf_pipe, df_train_features_numeric, df_test_features_numeric, df_train_target, df_test_target, "Rand
results.append(eval_model(gb_pipe, df_train_features_numeric, df_test_features_numeric, df_train_target, df_test_target, "Grad

pd.DataFrame(results)
```

Out[34]:

	Model	Accuracy	Macro F1	Macro Precision	Macro Recall
0	SKL Decision Tree	0.603448	0.604501	0.601923	0.633690
1	Logistic Regression	0.931034	0.920775	0.935417	0.908675
2	Random Forest	0.775862	0.765012	0.769176	0.763993
3	Gradient Boosting	0.758621	0.749035	0.744048	0.761378

Classification Conclusions

Model Comparisons

- **First Method: Building my own decision tree** This model demonstrated a mediocre performance on both the train and test sets, with an accuracy between 0.74 and 0.80. The method used to determine the node splits was based on observing the graphs and identifying correlations between the different features and their respective classes. We can see that the model had a slight tendency to classify runners as 'Advanced,' which is reflected in its low precision score for that class (it classified 5 'Intermediate' runners as 'Advanced'). This may have occurred due to the similarity of some features in the data (e.g., Average HR, Distance) between the classes.

- **Second Method: Scikit-learn (SKL) Decision Tree** This model demonstrated the poorest performance on the test set and showed signs of being overfitted to the training data (train accuracy: 0.82, test accuracy: 0.60). Furthermore, the model was effective at identifying 'Beginner' runners but performed poorly on the other classes. We can infer that this occurred because the relatively low pace of the 'Beginner' class allowed the model to find a 'pure' condition to separate the majority of 'Beginner' runners from the rest of the data.

- **Third Method: Logistic Regression Model** This model demonstrated the best performance among all models tested. The key features in the regression equation were 'Calculated Pace' and 'Distance,' which led to excellent model performance across all metrics (including accuracy, recall, and precision). Additionally, the model performed consistently well across all classes.

- **Fourth Method: Random Forest** The proposed model demonstrated a marginal improvement in performance over a standard decision tree. The slightly reduced performance observed on the training set is a key indicator that the model has avoided overfitting. This model,

consistent with the behavior of other decision tree algorithms in this project, achieved its highest performance metrics for the Beginner class.

- **Fifth Method: Gradient Boosting** The proposed model exhibited performance metrics nearly identical to those of the Random Forest model. It is important to note that the sample size is small, which means every single misclassification has a significant impact on the overall metrics. This model appears to rely heavily on the feature 'Calculated Pace.' Both models achieved their highest performance metrics for the Beginner class. Additionally, an analysis of the 'Advanced' class performance suggests the model may have acquired a "wrong" heuristic during training that led to poorer results on the test set, specifically in its ability to accurately classify this group.

General Conclusions

- Overall, 'Calculated Pace' had the biggest influence on the classification results, which makes sense since the GRPS score (which determined the label of each runner) was based on the runner's pace (better runners will generally have a better pace).
- Although I initially thought it would have a more significant influence on the different models, 'Average Heart Rate' did not appear to be a significant factor in determining the class of a runner. With that said, it has been the second most important feature in Random Forest and Gradient Boosting.

Sensitivity Analysis: Removing Target Leakage (Calculated Pace)

Purpose -

- In this section, we intentionally omit Calculated Pace from the feature set in order to compare model performance without this variable, which appears to have a substantial impact on predictive accuracy.
- We will evaluate the non-linear models Random Forest and Gradient Boosting, as the relationship between heart rate, distance, and runner tier is complex and likely non-linear. For example, a low heart rate at a short distance may indicate an intermediate runner, whereas a similarly low heart rate at a long distance may indicate an advanced runner. I expect that logistic regression may struggle to capture these interactions unless they are explicitly engineered.

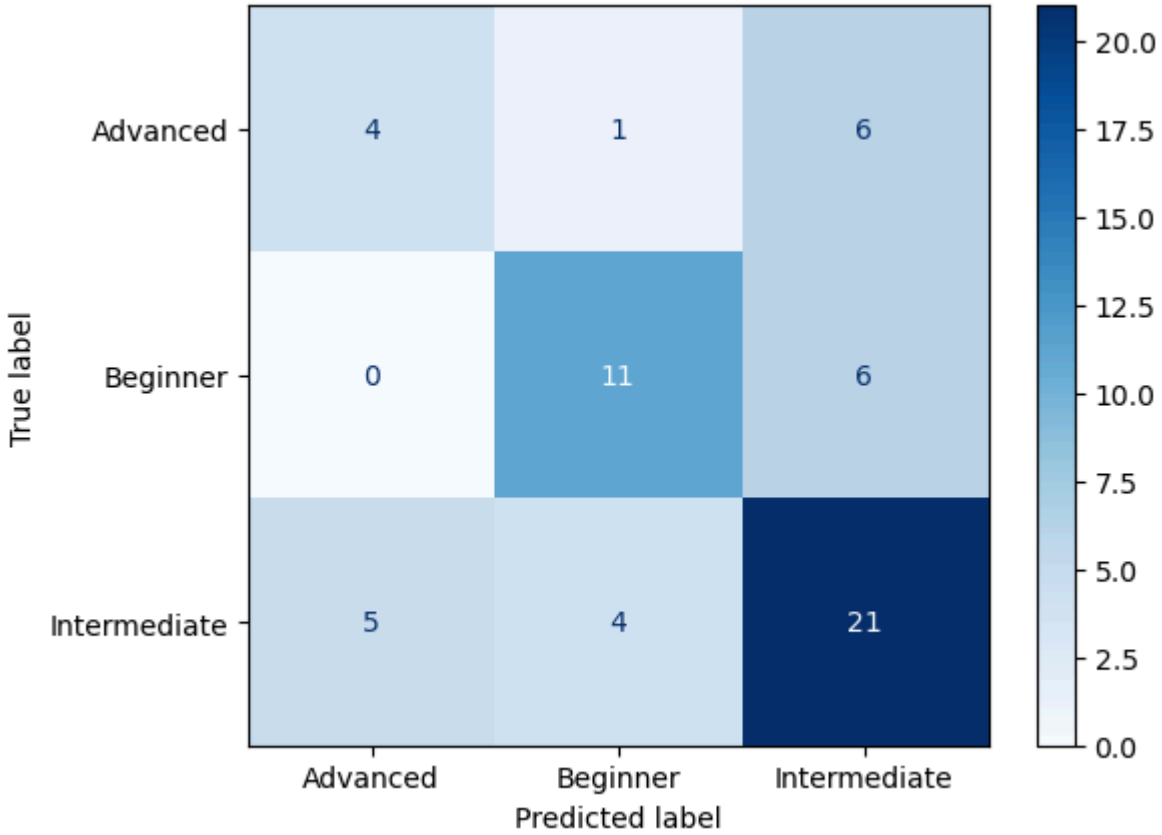
In [35]:
```
df_train_features_numeric_no_pace = df_train_features_numeric.drop(columns = "Median Calculated Pace")
df_test_features_numeric_no_pace = df_test_features_numeric.drop(columns = "Median Calculated Pace")
print(df_train_features_numeric_no_pace.head(6))
```

	Median Avg HR	Median Distance	Median Time
139	129.7	17.01500	97.916667
25	124.0	9.27230	57.275000
82	136.4	5.73085	34.975000
144	131.9	1.32600	7.066667
66	141.7	5.00720	26.266667
248	120.7	2.77790	15.900000

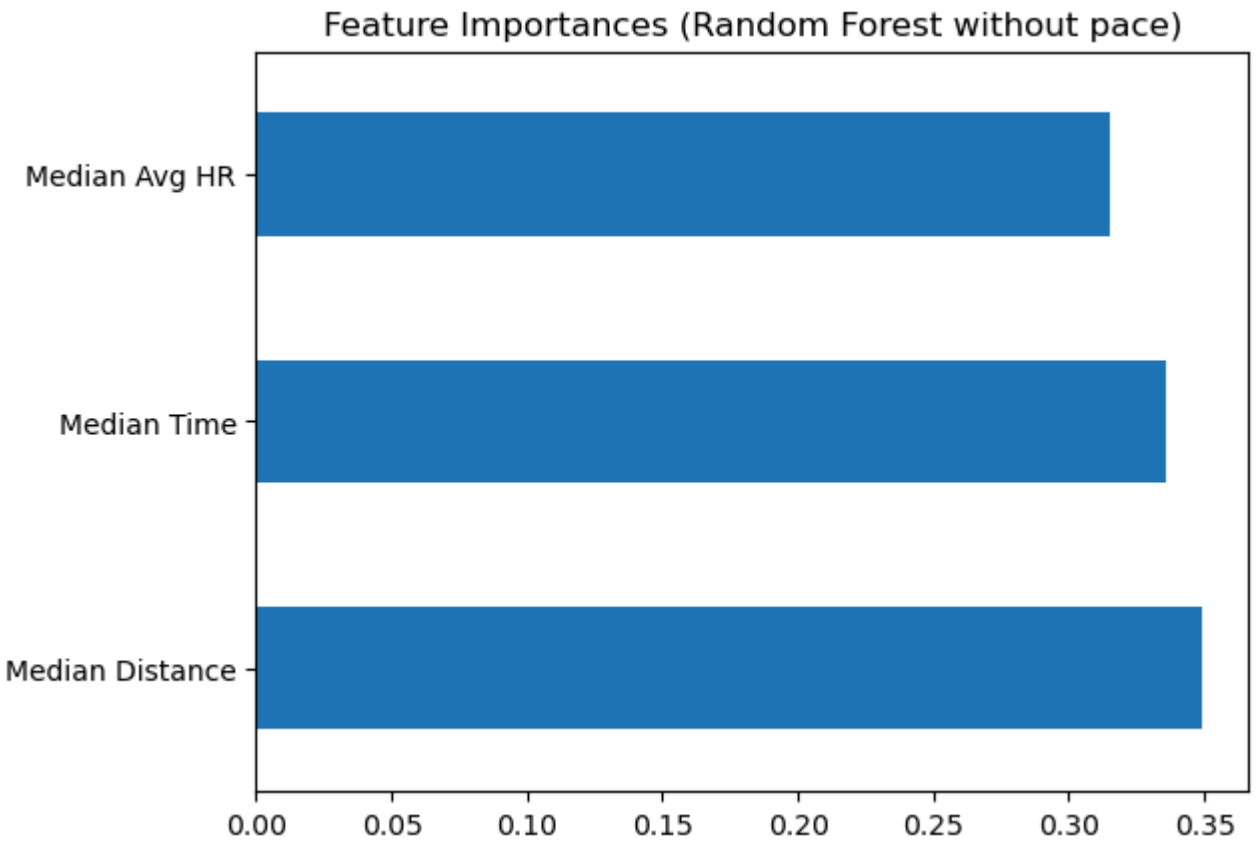
In [36]:
```
# Random forest Pipeline
features = final_df_model.drop(columns="Median Calculated Pace").select_dtypes(include=[int, float])
feature_cols = features.columns.tolist()
rf_pipe_no_pace = Pipeline([
    ("scaler", StandardScaler()),
    ("rf", RandomForestClassifier(
        n_estimators=200, max_depth=None, random_state=42
    ))
])

cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
y_pred_cv_no_pace = cross_val_predict(rf_pipe_no_pace, df_train_features_numeric_no_pace, df_train_target, cv=cv)
print("CV classification report - Random Forest:")
print(classification_report(df_train_target, y_pred_cv_no_pace, digits=3))

rf_pipe_no_pace.fit(df_train_features_numeric_no_pace, df_train_target)
y_pred_test_no_pace = rf_pipe_no_pace.predict(df_test_features_numeric_no_pace)
print("Test results classification report - Random Forest:")
print(classification_report(df_test_target, y_pred_test_no_pace, digits=3))

train_accuracy = accuracy_score(df_train_target, y_pred_cv_no_pace)
test_accuracy = accuracy_score(df_test_target, y_pred_test_no_pace)

results.append(eval_model(rf_pipe_no_pace, df_train_features_numeric_no_pace, df_test_features_numeric_no_pace, df_train_targe
```

```
CV classification report - Random Forest:
              precision    recall  f1-score   support

    Advanced      0.548     0.362     0.436        47
    Beginner      0.621     0.529     0.571        68
Intermediate      0.597     0.735     0.659       113

    accuracy                          0.596       228
   macro avg      0.589     0.542     0.555       228
weighted avg      0.594     0.596     0.587       228

Test results classification report - Random Forest:
              precision    recall  f1-score   support

    Advanced      0.444     0.364     0.400        11
    Beginner      0.688     0.647     0.667        17
Intermediate      0.636     0.700     0.667        30

    accuracy                          0.621        58
   macro avg      0.589     0.570     0.578        58
weighted avg      0.615     0.621     0.616        58
```

In [37]:
```
cm = confusion_matrix(df_test_target, y_pred_test_no_pace, labels=rf_pipe_no_pace.classes_)
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=rf_pipe_no_pace.classes_)
disp.plot(cmap="Blues")
plt.show()
```



In [38]:
```
importances = rf_pipe_no_pace.named_steps["rf"].feature_importances_
feat_imp = pd.Series(importances, index=feature_cols).sort_values(ascending=False)
feat_imp.plot(kind="barh")
plt.title("Feature Importances (Random Forest without pace)")
plt.show()
```



In [39]:
```
#Gradient boosting
gb_pipe_no_pace = Pipeline([
    ("scaler", StandardScaler()),
    ("gb", GradientBoostingClassifier(
        n_estimators=300, learning_rate=0.05, max_depth=3, random_state=42
    ))
])

y_pred_cv_no_pace = cross_val_predict(gb_pipe_no_pace, df_train_features_numeric_no_pace, df_train_target, cv=cv)
print("CV results - Classification report - Gradient Boosting Without Pace:")
print(classification_report(df_train_target, y_pred_cv_no_pace, digits=3))

gb_pipe_no_pace.fit(df_train_features_numeric_no_pace, df_train_target)
y_pred_test_no_pace = gb_pipe_no_pace.predict(df_test_features_numeric_no_pace)
```

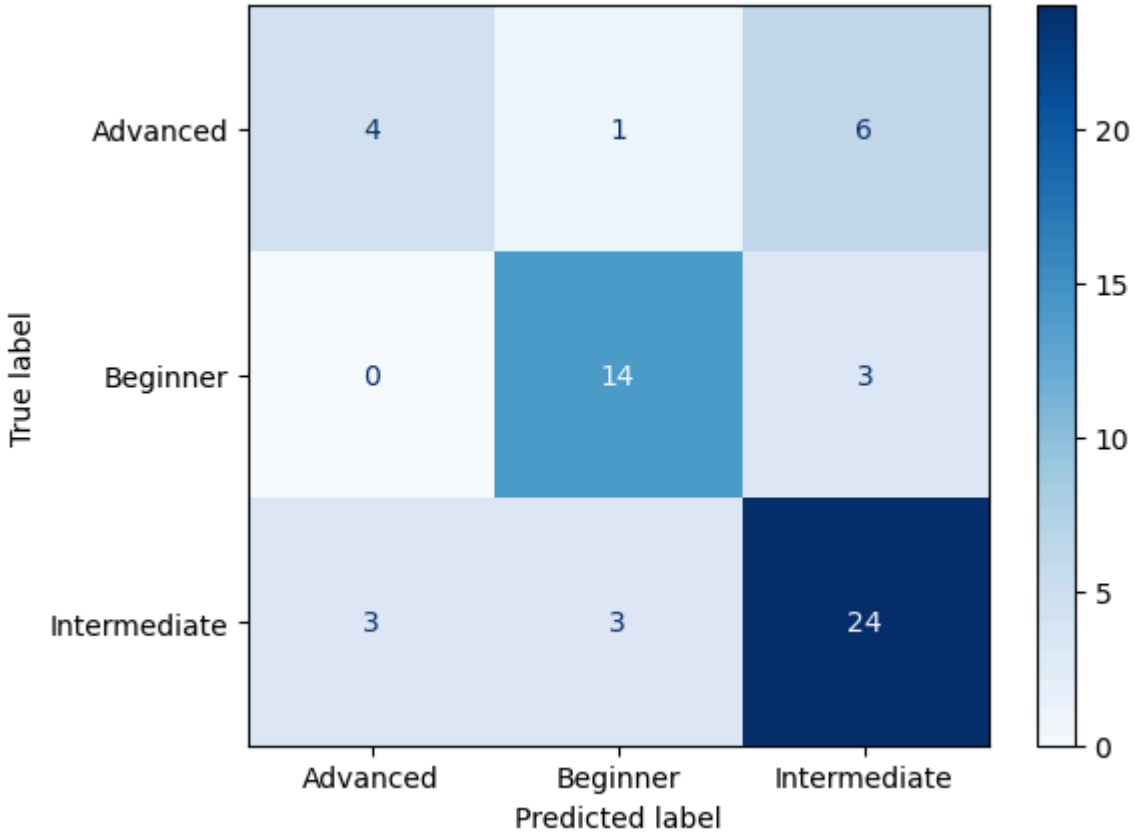
```
print("Test results - Classification report - Gradient Boosting Without Pace:")
print(classification_report(df_test_target, y_pred_test_no_pace, digits=3))

train_accuracy = accuracy_score(df_train_target, y_pred_cv_no_pace)
test_accuracy = accuracy_score(df_test_target, y_pred_test_no_pace)

results.append(eval_model(gb_pipe_no_pace, df_train_features_numeric_no_pace, df_test_features_numeric_no_pace, df_train_targe
```

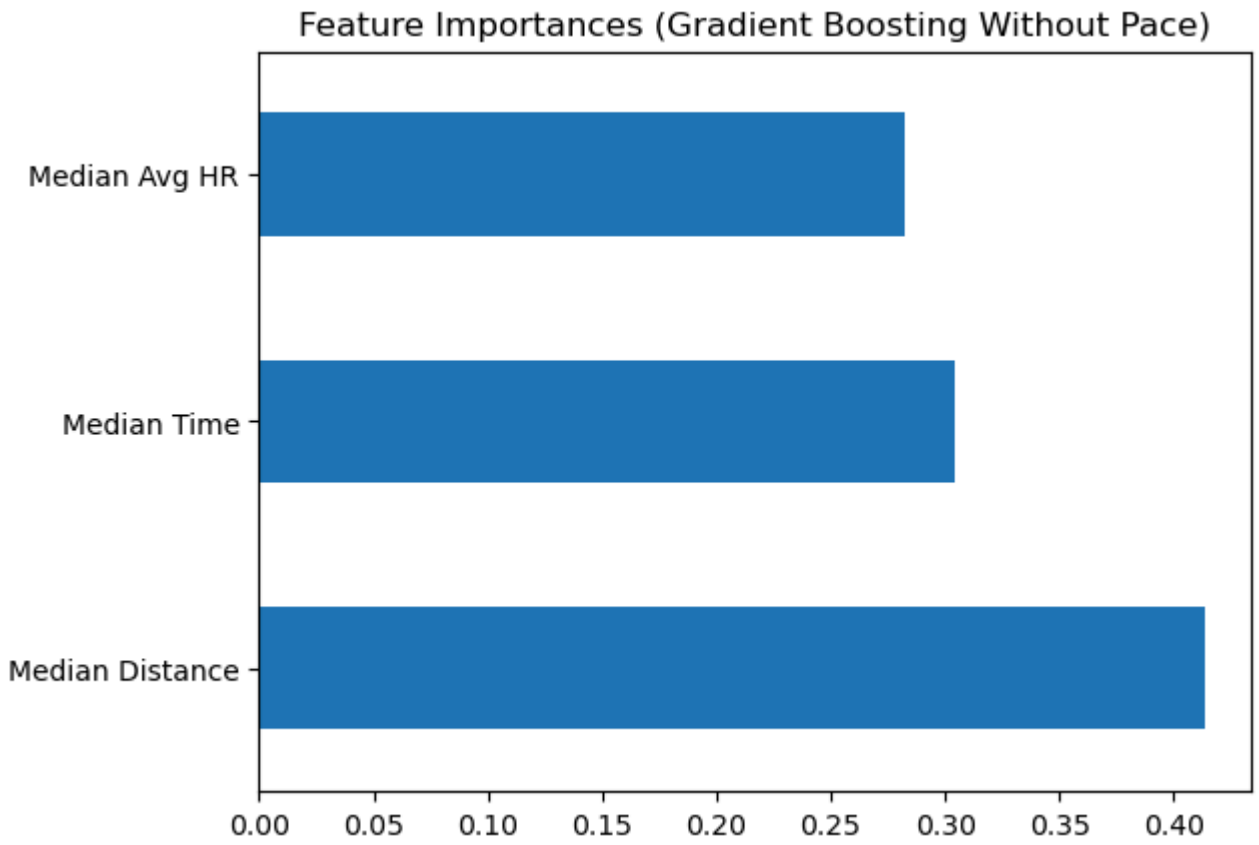
```
CV results - Classification report - Gradient Boosting Without Pace:
              precision    recall  f1-score   support

    Advanced      0.419     0.383     0.400        47
    Beginner      0.714     0.515     0.598        68
Intermediate      0.603     0.726     0.659       113

    accuracy                          0.592       228
   macro avg      0.579     0.541     0.552       228
weighted avg      0.598     0.592     0.587       228

Test results - Classification report - Gradient Boosting Without Pace:
              precision    recall  f1-score   support

    Advanced      0.571     0.364     0.444        11
    Beginner      0.778     0.824     0.800        17
Intermediate      0.727     0.800     0.762        30

    accuracy                          0.724        58
   macro avg      0.692     0.662     0.669        58
weighted avg      0.713     0.724     0.713        58
```

In [40]:
```
print("Confusion matrix - Gradient Boosting Without Pace")
cm = confusion_matrix(df_test_target, y_pred_test_no_pace, labels=gb_pipe_no_pace.classes_)
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=gb_pipe_no_pace.classes_)
disp.plot(cmap="Blues")
plt.show()

importances = gb_pipe_no_pace.named_steps["gb"].feature_importances_
feat_imp = pd.Series(importances, index=feature_cols).sort_values(ascending=False)
feat_imp.plot(kind="barh")
```

```
plt.title("Feature Importances (Gradient Boosting Without Pace)")
plt.show()
```

```
Confusion matrix - Gradient Boosting Without Pace
```





In [41]:
```
results_df = pd.DataFrame(results).sort_values("Accuracy", ascending = False)

styled_results = (
    results_df.style
    .format({"Accuracy": "{:.3f}", "Macro F1": "{:.3f}", "Macro Precision":"{:.3f}", "Macro Recall":"{:.3f}"})
    .highlight_max("Accuracy", color="#ABEBC6")
    .highlight_max("Macro F1", color="#ABEBC6")
    .highlight_max("Macro Precision", color="#ABEBC6")
    .highlight_min("Accuracy", color="#F5B7B1")
    .highlight_min("Macro F1", color = "#F5B7B1")
    .highlight_min("Macro Precision", color = "#F5B7B1")
    .highlight_max("Macro Recall", color = "#ABEBC6")
    .highlight_min("Macro Recall", color = "#F5B7B1")
```

```
)
display(styled_results)
```

	Model	Accuracy	Macro F1	Macro Precision	Macro Recall
1	Logistic Regression	0.931	0.921	0.935	0.909
2	Random Forest	0.776	0.765	0.769	0.764
3	Gradient Boosting	0.759	0.749	0.744	0.761
5	Gradient Boosting Without Pace	0.724	0.669	0.692	0.662
4	Random Forest Without Pace	0.621	0.578	0.589	0.570
0	SKL Decision Tree	0.603	0.605	0.602	0.634

Conclusions – Comparison Between Models With and Without Calculated Pace

- The Random Forest model achieved an accuracy of 77% when trained with Calculated Pace, but this performance was influenced by data leakage. When the feature was removed, accuracy dropped to 62%.
- The Gradient Boosting model achieved 75% accuracy with Calculated Pace (also affected by leakage), and 72% accuracy when the feature was excluded.
- Overall, for a real-world application, the Gradient Boosting model without Calculated Pace is likely the preferred choice, as it remains robust without relying on leakage and is therefore more suitable for deployment.

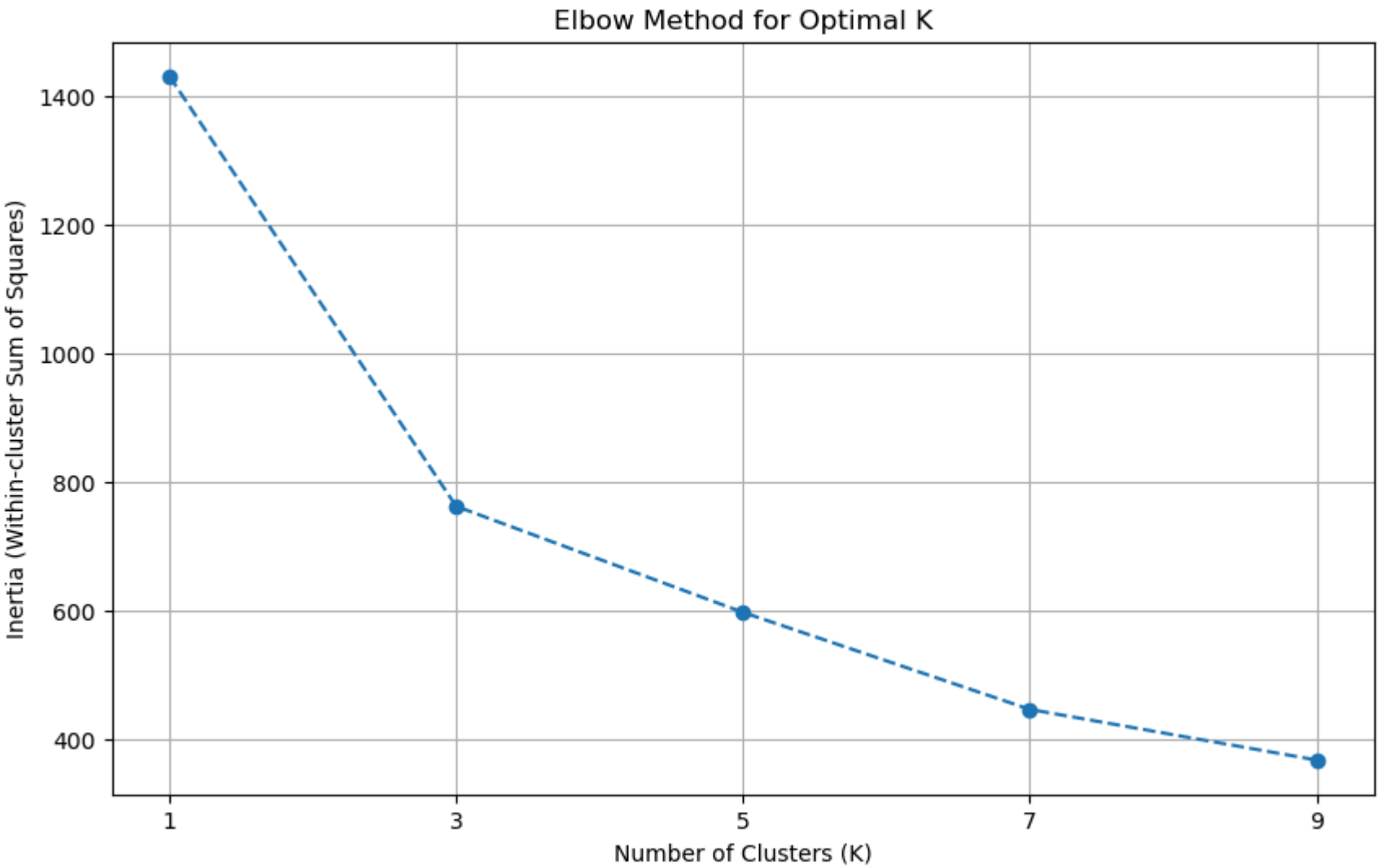
Clustering Method - using Kmeans clustering

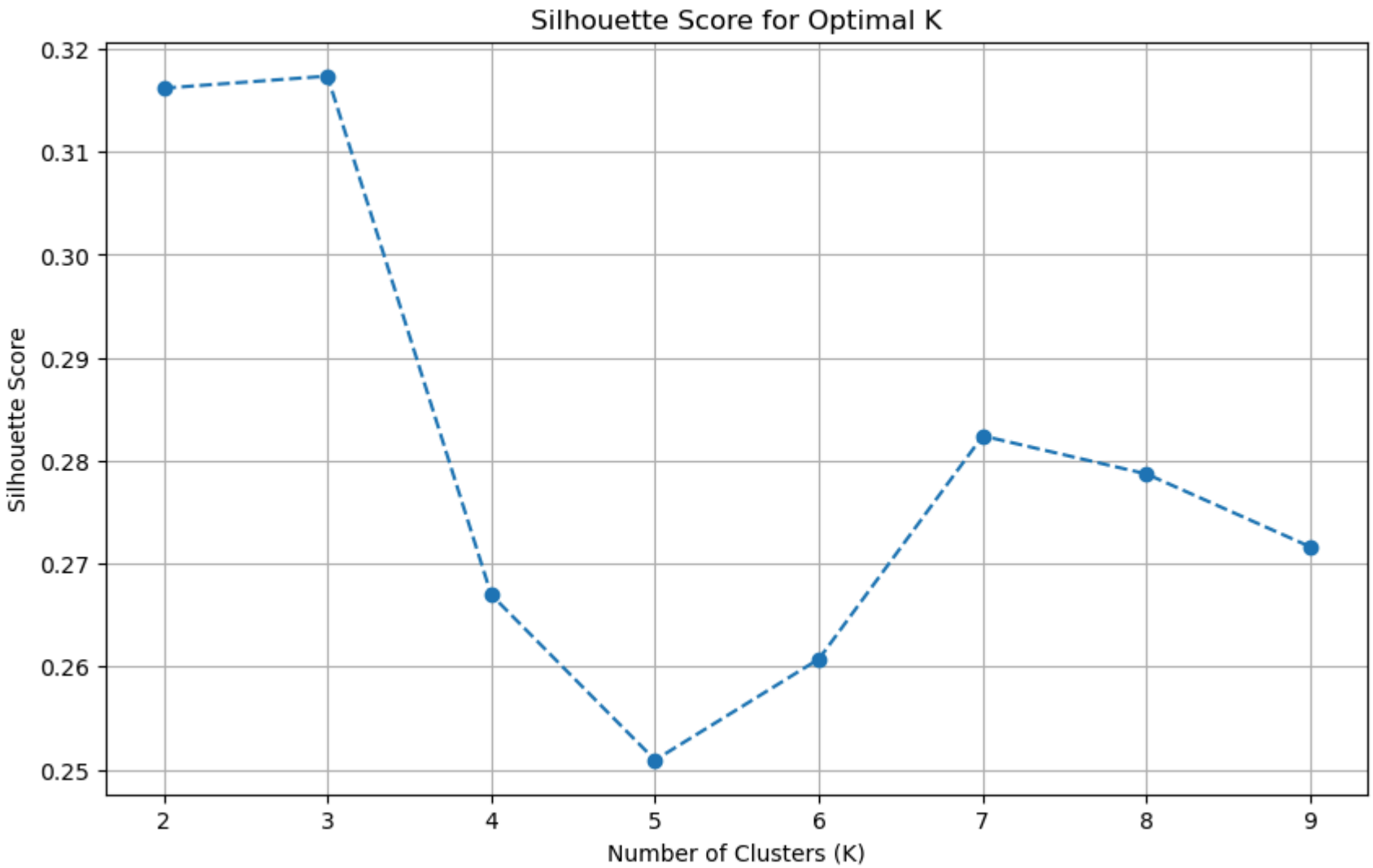
In [42]:
```
#Features Scaling
features = final_df.select_dtypes(include=[int, float])

scaler=StandardScaler()
normalized_features = scaler.fit_transform(features)
```

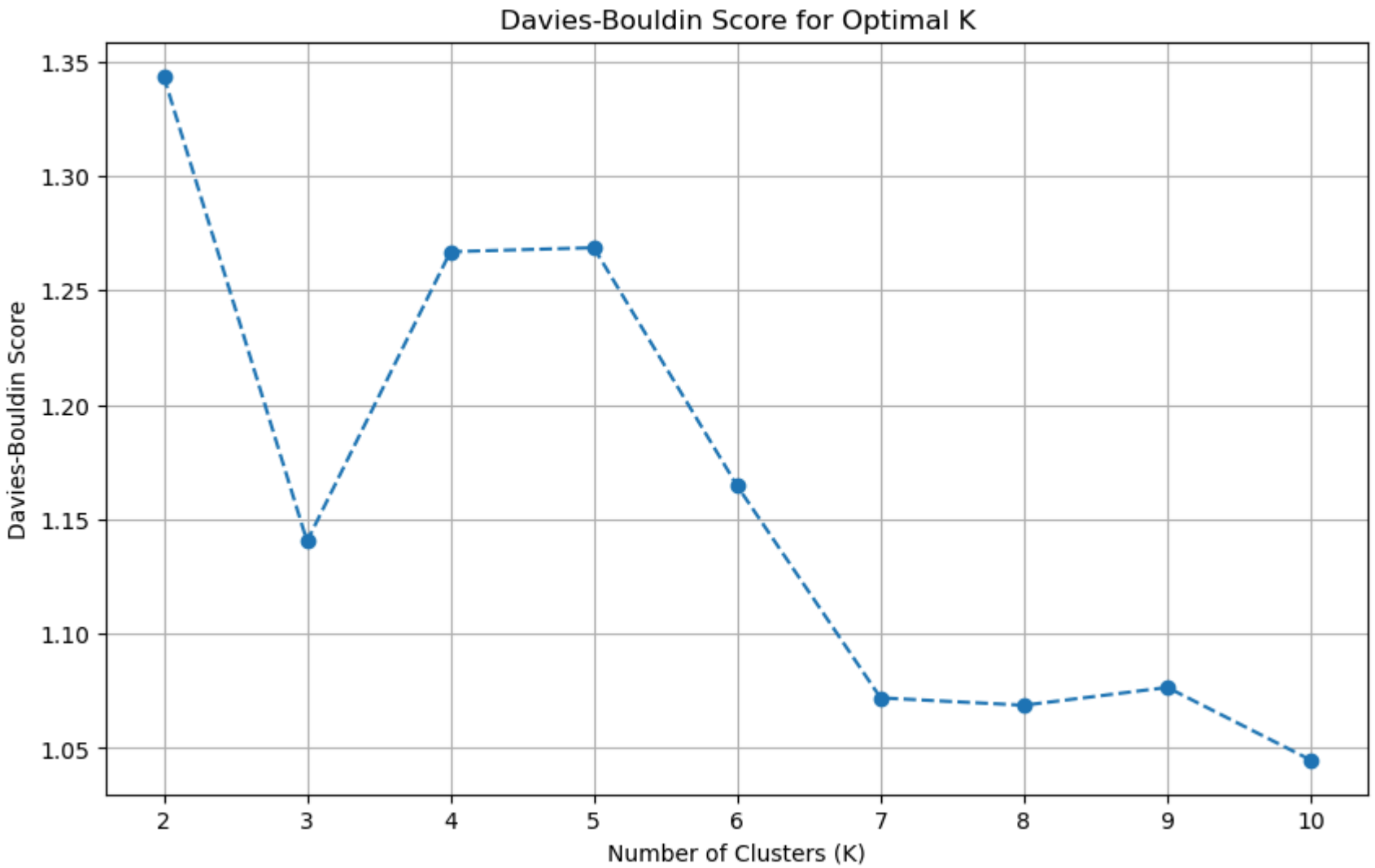
In [43]:
```
#finding the best K with Interia, Silhouette
inertia_values = []
max_k = 9
k_range = range(1, max_k + 1, 2)
```

```
for k in k_range:
    kmeans = KMeans(n_clusters=k, random_state=42, n_init='auto') #runs the algorithm multiple times with different random ini
    kmeans.fit(normalized_features)
    inertia_values.append(kmeans.inertia_)

plt.figure(figsize=(10, 6))
plt.plot(k_range, inertia_values, marker='o', linestyle='--')
plt.title('Elbow Method for Optimal K')
plt.xlabel('Number of Clusters (K)')
plt.ylabel('Inertia (Within-cluster Sum of Squares)')
plt.xticks(k_range)
plt.grid(True)
plt.show()

silhouette_scores = []
k_range_silhouette = range(2, max_k + 1)

for k in k_range_silhouette:
    kmeans = KMeans(n_clusters=k, random_state=42, n_init='auto')
    cluster_labels = kmeans.fit_predict(normalized_features)
    score = silhouette_score(normalized_features, cluster_labels)
    silhouette_scores.append(score)

plt.figure(figsize=(10, 6))
plt.plot(k_range_silhouette, silhouette_scores, marker='o', linestyle='--')
plt.title('Silhouette Score for Optimal K')
plt.xlabel('Number of Clusters (K)')
plt.ylabel('Silhouette Score')
plt.xticks(k_range_silhouette)
plt.grid(True)
plt.show()

db_scores = []
max_k = 10
k_range_db = range(2, max_k + 1)
for k in k_range_db:
    kmeans = KMeans(n_clusters=k, random_state=42, n_init='auto')
    cluster_labels = kmeans.fit_predict(normalized_features)
    score = davies_bouldin_score(normalized_features, cluster_labels)
    db_scores.append(score)
```

```
plt.figure(figsize=(10, 6))
plt.plot(k_range_db, db_scores, marker='o', linestyle='--')
plt.title('Davies-Bouldin Score for Optimal K')
plt.xlabel('Number of Clusters (K)')
plt.ylabel('Davies-Bouldin Score')
plt.xticks(k_range_db)
plt.grid(True)
plt.show()
```


Elbow Method for Optimal K
Inertia (Within-cluster Sum of Squares)
1400
1200
1000
800
600
400
1
3
5
7
9
Number of Clusters (K)


Silhouette Score for Optimal K
Silhouette Score
0.32
0.31
0.30
0.29
0.28
0.27
0.26
0.25
2
3
4
5
6
7
8
9
Number of Clusters (K)



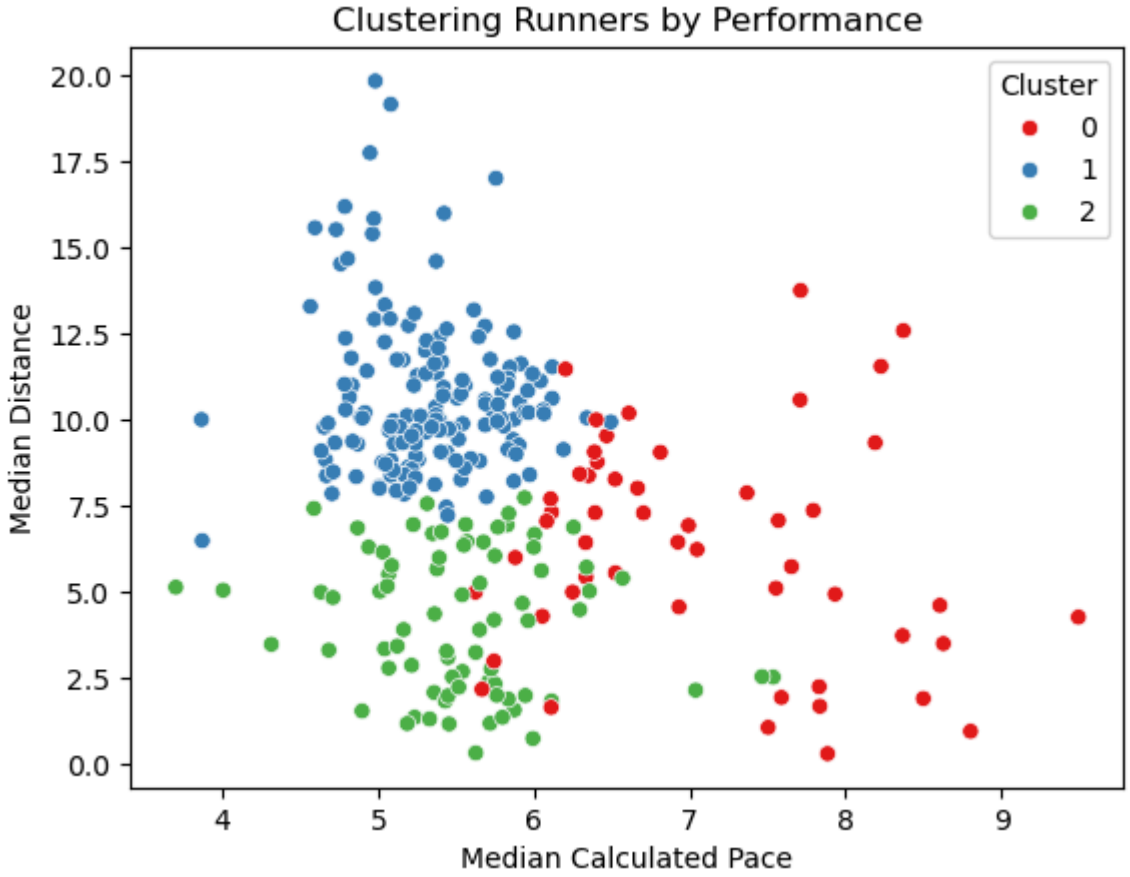
The Elbow Method - A method which helps us to choose the best number of centroids (k) in our model. The core idea behind the elbow method is that as you increase the number of clusters, the data points in each cluster will become closer to their respective cluster centroids. This will decrease the "inertia," or the sum of squared distances, for all the clusters At some point, adding another cluster will not significantly reduce the inertia. This is the "elbow" of the curve, where the rate of decrease in inertia slows down dramatically. This "elbow" point is considered to be the optimal number of clusters. The goal of the Elbow Method is to find the point where this decrease in inertia is no longer worth the added complexity of another cluster.

When using the Silhouette score, the optimal number of clusters (K) is identified by the highest score, as this indicates well-separated and compact clusters.

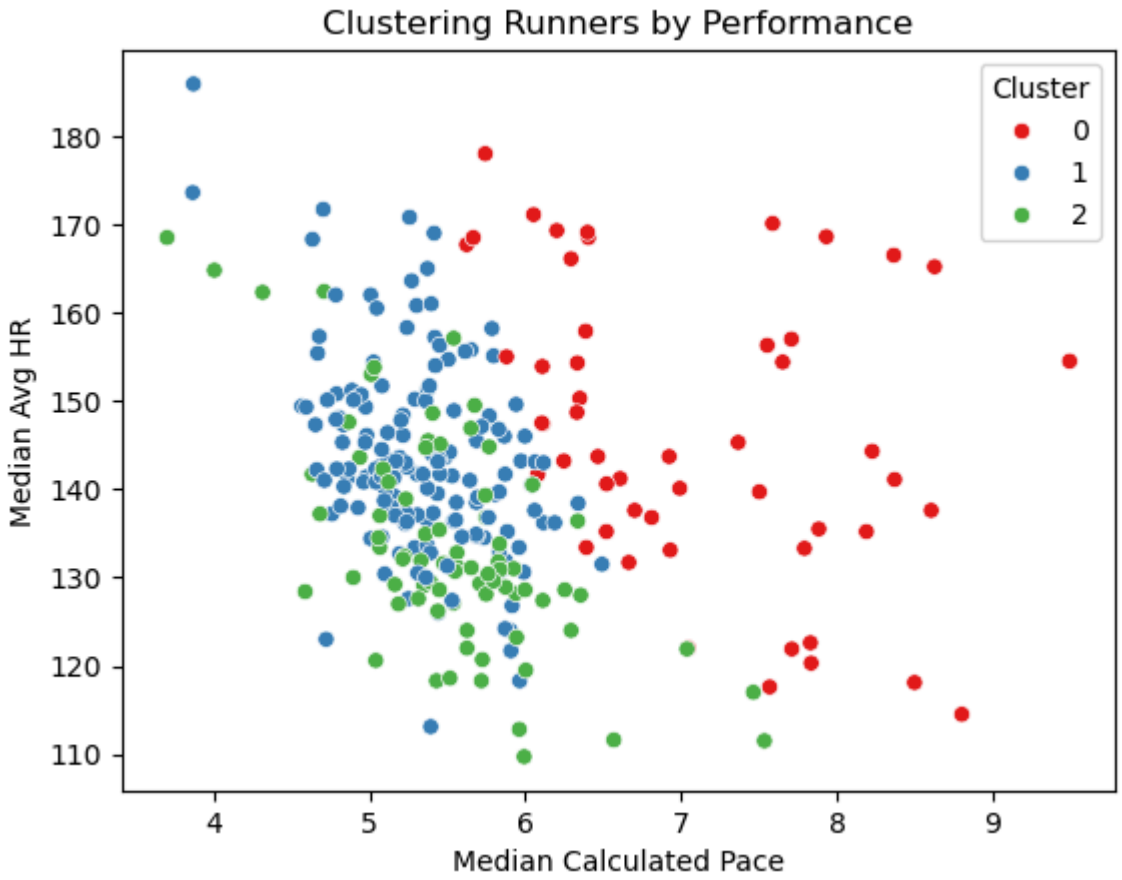
The Davies-Bouldin score graph shows a general trend of decreasing values as K increases. However, it can be observed that the value decreases most significantly at K=3. This value represents a balance between a low Davies-Bouldin score and the need to avoid overfitting with an excessive number of clusters.

Based on these findings, K=3 is the optimal number of clusters. This is supported by two key pieces of evidence: it corresponds to the highest Silhouette score, and it is the point in the Elbow method where the rate of decrease in inertia begins to diminish, forming a clear "elbow."

In [44]:
```
kmeans = KMeans(n_clusters=3, random_state=42)
cluster_labels = kmeans.fit_predict(normalized_features)
final_df["Cluster"] =  cluster_labels
sns.scatterplot(x="Median Calculated Pace", y="Median Distance", hue="Cluster", data=final_df, palette="Set1")
plt.title("Clustering Runners by Performance")
plt.show()
```

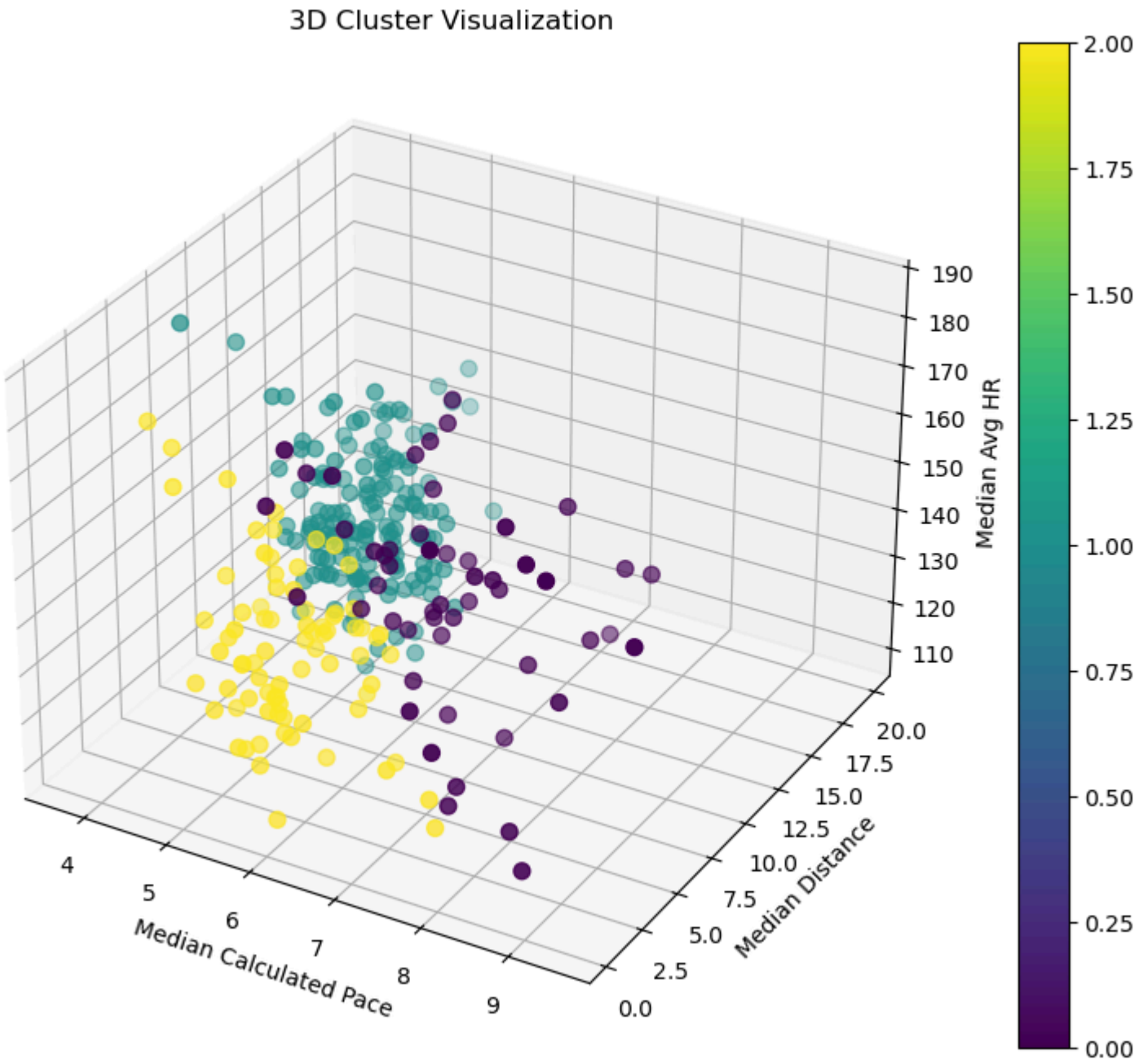


In [45]:
```
sns.scatterplot(x="Median Calculated Pace", y="Median Avg HR", hue="Cluster", data=final_df, palette="Set1")
plt.title("Clustering Runners by Performance")
plt.show()
```



In [46]:
```
optimal_k = 3

fig = plt.figure(figsize=(10, 8))
ax = fig.add_subplot(111, projection='3d')

sc = ax.scatter(final_df["Median Calculated Pace"], final_df['Median Distance'], final_df['Median Avg HR'], c=final_df['Cluste
plt.colorbar(sc)
ax.set_xlabel('Median Calculated Pace')
ax.set_ylabel('Median Distance')
ax.set_zlabel('Median Avg HR')
ax.set_title('3D Cluster Visualization')
```

```
plt.show()
```


3D Cluster Visualization
Median Calculated Pace
4
5
6
7
8
9
Median Distance
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
Median Avg HR
110
120
130
140
150
160
170
180
190
2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00

Kmeans Conclusions

- We can see that the clustering provided clusters with the next characteristics:

Cluster 0 - Low pace runners, distance is wide-spreaded (our 'Beginners')

Cluster 1 - Medium to high pace ruuners, longer distances (our 'Advanced')

Cluster 2 - Medium to high pace runners, short distances (our 'Intermediate')

- The analysis of the cluster's characteristics suggests that average heart rate was not a primary factor in the formation of the clusters. A consistent range of average heart rates is observed across all clusters, indicating a uniform distribution of this feature. This finding is reinforced by the fact that the majority of runners have an average heart rate between 130 and 165 BPM.

Example - Based on the clustering we've processed I belong to cluster 1 which makes me a medium to high pace runner for long distances

In [47]:
```
print(final_df.loc[final_df["Runner"] == "maor"])
```

```
    Runner  Median Avg HR  Median Distance  Median Time  \
281   maor          162.0             8.02    40.766667

     Median Calculated Pace  Median GRPS Score  Runner_Class  Cluster
281                5.006653          23.651118  Intermediate        1
```

Using ARI and V-measure to compare the clusters to our initial GRPS classes

In [48]:
```
class_mapping = {"Beginner": 0,"Intermediate": 2, "Advanced": 1}
final_df["Runner_Class_Numeric"] = final_df["Runner_Class"].map(class_mapping)
labels_true = final_df.loc[:, "Runner_Class_Numeric"]
labels_pred = final_df.loc[:, "Cluster"]
ari_score = adjusted_rand_score(labels_true, labels_pred)
print(f"Perfect match ARI score: {ari_score:.2f}")
v_score = v_measure_score(labels_true, labels_pred)
print(f"Perfect match V-measure score: {v_score:.2f}")
```

```
Perfect match ARI score: 0.20
Perfect match V-measure score: 0.28
```

We can see that the ARI is 0.2 and the V-measure is 0.28 which aren't great values. It may show that the Kmeans model found patterns in our data which are different from the way we have decided to label our runner. It also can happen because the features used for clustering may not be the most significant ones for distinguishing between the runner classes as we defined them.

Final conclusions

In this project I explored the problem of classifying runners into performance tiers using personal running data, common runing metrics (pace, distance, heart rate etc.) and a derived General Running Performance Score (GRPS) which is primarly based on Reigel formula.

I tested unsupervised clustering model (KMeans) and supervised models (Decision tree ,Logistic Regression, Random Forest, Gradient Boosting) and evaluated models using stratified cross-validation and a held-out test set.

The best performing classification models balanced precision and recall across the three classes, however, adjacent classes (Beginner vs Intermediate and Intermediate vs Advanced) remain the primary source of error, reflecting realistic overlap in runner abilities.

Feature-importance analysis showed that normalized pace was the most predictive feature to classify our runners based on the method we used to determine the labels (based on their GRPS score).

We also learned that our clustering model found diferrent patterns in the data than the labels we've had determined.

Overall, the project demonstrates a full end-to-end pipeline - data collection, feature engineering (GRPS metric), exploratory analysis, modeling based on multiple classification approaches and clustering , evaluation and has a touch to my daily areas of intrest. With a larger dataset and final polish on conclusions, it can stand out as a solid applied Data Science project.