Running Performance Classification Project

1. Problem Statement -
- Objective: Classify runners into perforance 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, than use the running metric chosen 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 positives 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 - 2 X Precision X Recall / (Precision + Recall)
- **Decision Tree metrics:**
  - Gini - Measures the impurity of data within a decision tree node when 0 means that the node is pure (contains only elements of a single class) and Higher values means that 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**
  - Interia - The sum of the squared distances between each data point and the centroid of its assigned cluster. It measures how 'tight' or 'compact' the class is. A lower interia 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 - The 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) when 1 means that the data point is well-matched to its own cluster and well-separated from neighboring clusters, 0 means that the data point is on or very close to the decision boundary between two clusters and -1 means that 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 it shows that the clusters are dense and well seperated.
- **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) - It 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 assigment 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.

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

In [2]:
```python
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
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,acc
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay, classifica
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.cluster import KMeans
from sklearn.preprocessing import QuantileTransformer,PowerTransformer,RobustSca
from sklearn.metrics import silhouette_score, davies_bouldin_score, adjusted_ran
from sklearn.ensemble import RandomForestClassifier, GradientBoostingClassifier


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

```
✅ All libraries are working!
```

In [3]:

```
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)
```

```
Path to dataset files: C:\Users\maors\.cache\kagglehub\datasets\olegoaer\running-
races-strava\versions\1
Path to dataset files: C:\Users\maors\.cache\kagglehub\datasets\purpleyupi\strava
-data\versions\2
```

In [4]:

```
## 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
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
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)
```

In [8]:

```
maor = pd.read_csv("datamaor's_data.csv") #C:\\Users\\maors\\Maor's studies\\dat
ely = pd.read_excel("C:/Users/maors/Maor's studies/datasets/Ely's_data(no_HR).xl
shapira = pd.read_excel("C:/Users/maors/Maor's studies/datasets/shapira's_data.x
someone = pd.read_csv("C:/Users/maors/Maor's studies/datasets/activity_log.csv")
franco = pd.read_excel("C:/Users/maors/Maor's studies/datasets/franco's_data.xls
danieli = pd.read_excel("C:/Users/maors/Maor's studies/datasets/ido's_data.xlsx"
ehud = pd.read_excel("C:/Users/maors/Maor's studies/datasets/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(shapira.head())
```

```
---------------------------------------------------------------------------
FileNotFoundError                         Traceback (most recent call last)
Cell In[8], line 1
----> 1 maor = pd.read_csv("maor's_data.csv") #C:\\Users\\maors\\Maor's studies
\\datasets\\
      2 ely = pd.read_excel("C:/Users/maors/Maor's studies/datasets/Ely's_data(no
_HR).xlsx")
      3 shapira = pd.read_excel("C:/Users/maors/Maor's studies/datasets/shapira's
_data.xlsx")

File ~\anaconda3\Lib\site-packages\pandas\io\parsers\readers.py:1026, in read_csv
(filepath_or_buffer, sep, delimiter, header, names, index_col, usecols, dtype, en
gine, converters, true_values, false_values, skipinitialspace, skiprows, skipfoot
er, nrows, na_values, keep_default_na, na_filter, verbose, skip_blank_lines, pars
e_dates, infer_datetime_format, keep_date_col, date_parser, date_format, dayfirs
t, cache_dates, iterator, chunksize, compression, thousands, decimal, linetermina
tor, quotechar, quoting, doublequote, escapechar, comment, encoding, encoding_err
ors, dialect, on_bad_lines, delim_whitespace, low_memory, memory_map, float_preci
sion, storage_options, dtype_backend)
   1013 kwds_defaults = _refine_defaults_read(
   1014     dialect,
   1015     delimiter,
   (...)
   1022     dtype_backend=dtype_backend,
   1023 )
   1024 kwds.update(kwds_defaults)
-> 1026 return _read(filepath_or_buffer, kwds)

File ~\anaconda3\Lib\site-packages\pandas\io\parsers\readers.py:620, in _read(fil
epath_or_buffer, kwds)
    617 _validate_names(kwds.get("names", None))
    619 # Create the parser.
--> 620 parser = TextFileReader(filepath_or_buffer, **kwds)
    622 if chunksize or iterator:
    623     return parser

File ~\anaconda3\Lib\site-packages\pandas\io\parsers\readers.py:1620, in TextFile
Reader.__init__(self, f, engine, **kwds)
   1617     self.options["has_index_names"] = kwds["has_index_names"]
   1619 self.handles: IOHandles | None = None
-> 1620 self._engine = self._make_engine(f, self.engine)

File ~\anaconda3\Lib\site-packages\pandas\io\parsers\readers.py:1880, in TextFile
Reader._make_engine(self, f, engine)
   1878     if "b" not in mode:
   1879         mode += "b"
-> 1880 self.handles = get_handle(
   1881     f,
   1882     mode,
   1883     encoding=self.options.get("encoding", None),
   1884     compression=self.options.get("compression", None),
   1885     memory_map=self.options.get("memory_map", False),
   1886     is_text=is_text,
   1887     errors=self.options.get("encoding_errors", "strict"),
   1888     storage_options=self.options.get("storage_options", None),
   1889 )
   1890 assert self.handles is not None
   1891 f = self.handles.handle

File ~\anaconda3\Lib\site-packages\pandas\io\common.py:873, in get_handle(path_or
_buf, mode, encoding, compression, memory_map, is_text, errors, storage_options)
    868 elif isinstance(handle, str):
    869     # Check whether the filename is to be opened in binary mode.
    870     # Binary mode does not support 'encoding' and 'newline'.
    871     if ioargs.encoding and "b" not in ioargs.mode:
    872         # Encoding
--> 873         handle = open(
    874             handle,
    875             ioargs.mode,
    876             encoding=ioargs.encoding,
    877             errors=errors,
    878             newline="",
    879         )
    880     else:
    881         # Binary mode
    882         handle = open(handle, ioargs.mode)

FileNotFoundError: [Errno 2] No such file or directory: "maor's_data.csv"
```

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 Avg HR.

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 distance)

This effectively accounts for the factor of distance and allows you to compare the "effort" of a 10K race to an easy long run/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 benchmark HR (all the participants are between the age 23-26)

In [ ]:
```
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 ti
    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 [ ]:
```
#GRPS calculation
for i in range(len(dfs)):
    df = dfs[i]
    df["T2"] = df["Time"] * (10/df["Distance"]) ** 1.06
    df["Normalized_HR"] = df["Avg HR"] / 195
    df["GRPS"] = ((10 / df["T2"]) / df["Normalized_HR"]) * 100
    dfs[i] = df
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(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}")
```

In [ ]:
```
#Creatimg a median run Data frame with synthetic 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.

In [ ]:
```
strava_df = pd.read_excel("C:\\Users\\maors\\Maor's studies\\datasets\\raw-data-

# If the data is stored in a single column as semicolon-separated strings, split
if strava_df.shape[1] == 1 and strava_df.columns[0] == 'athlete;gender;timestamp
```

```
    strava_df = strava_df.iloc[:, 0].str.split(";", expand=True) # Split the sin
    strava_df.columns = ["athlete", "gender", "timestamp", "distance (m)","elaps

strava_df["timestamp"] = pd.to_datetime(strava_df["timestamp"], dayfirst=True)
strava_df["distance (m)"] = pd.to_numeric(strava_df["distance (m)"], errors='coe
strava_df["elapsed time (s)"] = pd.to_numeric(strava_df["elapsed time (s)"], err
strava_df["elevation gain (m)"] = pd.to_numeric(strava_df["elevation gain (m)"],
strava_df["average heart rate (bpm)"] = pd.to_numeric(strava_df["average heart r
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)", "timesta
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", "Dis
median_strava_runs["Avg HR"] = strava_df.groupby("Runner")["Avg HR"].transform(l
median_strava_runs["T2"] = median_strava_runs["Time"] * (10 / median_strava_runs
median_strava_runs["Normalized_HR"] = median_strava_runs["Avg HR"] / 195
median_strava_runs["GRPS score"] = (10 / median_strava_runs["T2"]) / median_stra
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 [ ]:
```
strava2_df = pd.read_csv("C:\\Users\\maors\\.cache\\kagglehub\\datasets\\purpley
strava2_df.drop(columns = ["kudos_count","start_date_local","type","elev_high",
#strava2_df["distance"] = pd.to_numeric(strava2_df["distance"], errors='coerce')
strava2_df["moving_time"] = strava2_df["moving_time"].apply(time_to_minutes).ast
strava2_df["average_heartrate"] = pd.to_numeric(strava2_df["average_heartrate"],
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", "D
median_strava2_runs = median_strava2_runs.dropna(subset=["Avg HR"])
median_strava2_runs["T2"] = median_strava2_runs["Time"] * (10 / median_strava2_r
median_strava2_runs["Normalized_HR"] = median_strava2_runs["Avg HR"] / 195
median_strava2_runs["GRPS score"] = (10 / median_strava2_runs["T2"]) / median_st
median_strava2_runs.drop(columns = ["Normalized_HR","T2"], inplace = True)
median_strava2_runs = median_strava2_runs.loc[median_strava2_runs['GRPS score']
median_strava2_runs = median_strava2_runs.loc[median_strava2_runs['Calculated Pa
#print(median_strava2_runs.sample(7))
#print(median_strava2_runs.info())
#print(median_strava2_runs.describe())
```

In [ ]:
```
final_strava = pd.concat([median_strava_runs, median_strava2_runs], ignore_index
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"}, inpla
final_strava.rename(columns={"GRPS score": "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.
#print(final_df.tail(7))
#print(final_df.info())
```

In [ ]:
```
final_df = final_df.loc[final_df['Median Time'] < 150]
final_df = final_df.loc[final_df['Median Calculated Pace'] < 9.5]
print(final_df.info())
```

In [ ]:
```
numeric_cols = final_df.select_dtypes(include=np.number)
corr_mat = numeric_cols.corr()
print(sns.heatmap(corr_mat, annot=True, cmap="coolwarm", fmt=".2f", linewidths=0
```

In [ ]:
```
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()

for i, col in enumerate(numeric_cols):
    ax = axes[i]
    data = [final_df[final_df['Runner_Class'] == cls][col] for cls in runner_cla
    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()
```

In [ ]:
```
final_df_model = final_df.drop(columns = ["Median GRPS Score"])
train_split, test_split = train_test_split(final_df_model, test_size = 0.2, rand
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: Distance, Calculated pace, avg HR and GRPS vs Runner Class where (1 = Beginner, 2 = Intermediate, 3 = Advanced)

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

In [ ]:
```
#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 H
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 [ ]:
```
#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 P
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 [ ]:
```
#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
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()
```

1st classification method - Manual Desicion Tree

We can infer from the data that there is a clear seperation between pace to the runner class (this might be a problem because it doesn't consider that marathon runners should run slower than 5K) and also that akk the runners who runs less than 8 km (on average) are beginners, with the remainig data (pace between 4.7 to 5.6) I chose to use Disatnce and the Avg HR

In [ ]:
```
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 [ ]:
```
#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)
```

1st Method Results

In [ ]:
```
print(classification_report(df_test_target, res_test))
```

In [ ]:
```
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=labe
disp.plot(cmap=plt.cm.Blues)
plt.show()
```

2nd Classification Method - SKL Decision Tree

Using the decision tree built in function of SKL

In [ ]:
```
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)
```

2nd Method Results

In [ ]:
```
print(classification_report(df_test_target, y_predict_test_sklearn))
print(classification_report(df_train_target, y_predict_train_sklearn))
```

In [ ]:
```
feature_names = ["Median Time", "Median Distance", "Median Avg HR", "Median Calc
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 [ ]:
```
cm_DT_func = confusion_matrix(df_test_target, y_predict_test_sklearn, labels=sk_
disp = ConfusionMatrixDisplay(confusion_matrix=cm_DT_func, display_labels=sk_tre
disp.plot(cmap=plt.cm.Blues)
plt.show()
```

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 [ ]:
```
#df_train_features_numeric_LR = df_train_features_numeric.drop(columns = ["Media
#df_test_features_numeric_LR = df_test_features_numeric.drop(columns = ["Median
pipe = Pipeline([('scaler', StandardScaler()),('logreg', LogisticRegressionCV(ma
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, 'Coef
print(coefficients)
print("Intercept:", pipe[1].intercept_)
```

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

3rd Method Results

In [ ]:
```
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

train_roc_auc = roc_auc_score(df_train_target, train_pred_proba, multi_class='ov
test_roc_auc = roc_auc_score(df_test_target, test_pred_proba, multi_class='ovr')

train_precision = precision_score(df_train_target, train_pred, average='weighted
test_precision = precision_score(df_test_target, test_pred, average='weighted')

train_recall = recall_score(df_train_target, train_pred, average='weighted')
test_recall = recall_score(df_test_target, test_pred, average='weighted') #Recal

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)
```

In [ ]:
```
cm_LR = confusion_matrix(df_test_target, test_pred, labels=pipe.classes_)
disp = ConfusionMatrixDisplay(confusion_matrix=cm_LR, display_labels=pipe.classe
disp.plot(cmap=plt.cm.Blues)
plt.show()
```

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 [ ]:
```
# 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_targe
print("CV results:")
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:")
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)
```

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

In [ ]:
```
importances = rf_pipe.named_steps["rf"].feature_importances_
feat_imp = pd.Series(importances, index=feature_cols).sort_values(ascending=Fals
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 [ ]:
```
#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_targe
print("CV results (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 (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)
```

In [ ]:
```
cm = confusion_matrix(df_test_target, y_pred_test, labels=gb_pipe.classes_)
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=gb_pipe.classe
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=Fals
feat_imp.plot(kind="barh")
plt.title("Feature Importances (Gradient Boosting)")
plt.show()
```

In [ ]:
```
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_n
results.append(eval_model(pipe, df_train_features_numeric, df_test_features_nume
results.append(eval_model(rf_pipe, df_train_features_numeric, df_test_features_n
results.append(eval_model(gb_pipe, df_train_features_numeric, df_test_features_n
pd.DataFrame(results)
```

Classification conclusions

Model Comparsions

- 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.

Clustering Method - using Kmeans clustering

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

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

In [ ]:
```
#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 algo
    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()
```

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 [ ]:
```
OPM_NUM_THREADS=2
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",
plt.title("Clustering Runners by Performance")
plt.show()
```

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

In [ ]:
```
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'],
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()
```

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.

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

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

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

In [ ]:
```
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}")
```

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.