Temporal_Metrics_Random_Forest_PartII

December 7, 2023

1 Temporal Metrics: Quantifying Human Time Perception

Regis University Cammie Newmyer October 8, 2023 Updated: December 3, 2023

2 Purpose Statement: Quantifying Human Time Perception

Objective: To introduce an innovative approach to understanding and quantifying human time perception, bridging the realms of neurology, physics, and psychology to offer new insights into a long-standing question: How do humans perceive the passage of time?

The Importance of Understanding Human Time Perception: The way humans perceive time has far-reaching implications not only for individual subjective experiences but also for broader social and functional contexts. Our perception of time shapes our reactions, decisions, memories, and future anticipations. It influences our emotional state, the rhythm of our daily activities, and even our cultural narratives. From waiting for a bus to recollecting past experiences, our sense of time pervades every aspect of our lives. Thus, understanding it is pivotal for both enhancing personal well-being and addressing societal challenges.

A Novel Framework: The Distance = Rate x Time Paradigm in Time Perception: At first glance, the formula "distance = rate x time" seems exclusive to the physical realm, typically associated with motion. However, we postulate that it can be metaphorically applied to human time perception. Here, 'distance' doesn't signify a physical journey but represents the cognitive journey our brains undertake in a 24-hour span, as they process myriad stimuli and experiences.

Distance: The theoretical "distance" our brains travel is a cumulative representation of all brainwave activities in a typical day. This cognitive journey can vary significantly among individuals, especially when considering conditions or disorders that impact time perception.

Rate: This is characterized by the wavelengths and frequencies of standard brainwaves - delta, theta, alpha, low beta, mid beta, high beta, and gamma. Each type of wave represents different cognitive states and functions, from deep sleep to heightened alertness.

Time: Time, in this context, refers to the duration one experiences a particular brainwave activity within 24 hours.

The Weber-Fechner law, or Weber's law: A principle in experimental psychology proposed by Ernst Heinrich Weber in 1834. It states that the just-noticeable difference between two stimuli is proportional to the magnitude of the stimuli. That is, if you increase a stimulus (like brightness, weight, loudness, etc.), the amount of change required for us to notice this change becomes larger as the stimulus itself becomes larger. In the context of time perception, applying Weber's law would mean that our ability to distinguish between two durations would be based on a ratio or proportion rather than a fixed quantity.

Incorporating Weber's Time Constant and Myelination: The Weber time constant, derived by calculating the difference between the 'distance' covered by an average brain compared to those with altered time perception, offers a scalar measure of time perception variations. Furthermore, factoring in changes in myelination, which can influence the speed and efficiency of neural transmissions, adds depth to our understanding. An increase in myelination can speed up neural processes, possibly leading to altered time perception, while a decrease might have the opposite effect.

3 Data Description

Methodology: Harnessing AI-Powered Insights for Understanding Human Time Perception

Data Acquisition:

Theoretical Judgements: Our research began by tapping into the theoretical frameworks of neuroscience, psychology, and time perception. By leveraging these well-established theories, we constructed a foundational understanding upon which more specific and nuanced data points could be built.

Rapid Fire Questioning with AI ChatGPT-4: To delve deeper into the complexities of time perception, we engaged in extensive interactions with OpenAI's ChatGPT-4. This state-of-the-art AI model, trained on vast amounts of data, provided us with nuanced insights and knowledge gaps in existing research.

Why ChatGPT-4?

Access to Comprehensive Data: ChatGPT-4 has been trained on a multitude of research papers, articles, and databases. Its knowledge spans a wide array of fields, allowing us to extract valuable information on brainwave activity, associated tasks, and conditions influencing time perception.

Efficient Interaction: Rapid-fire questioning with ChatGPT-4 ensured that our data acquisition process was both thorough and efficient. By posing sequential questions and building upon the AI's responses, we were able to derive detailed and interconnected insights within a short time frame.

Dynamic Learning Approach: ChatGPT-4's ability to understand and respond contextually enabled a more organic, conversational approach to data gathering. This dynamic interaction often led to the revelation of unexpected but valuable insights.

Data Analysis:

Brainwave Activity and Associated Tasks: With the data acquired, we charted out a comprehensive mapping of different brainwave types (delta, theta, alpha, beta, gamma) against their associated cognitive tasks. This provided a clear picture of how various activities or states of being could influence time perception.

Alterations in Human Time Perception: Further, we analyzed the factors leading to alterations in time perception. Data indicated a range of influences from biological (e.g., neurochemical changes, myelination variations) to psychological (e.g., trauma, mindfulness practices).

Conclusion:

By adopting this multifaceted approach, we aim to shed light on the intricate mosaic of human time perception. Unveiling its mysteries could lead to therapeutic breakthroughs for disorders affecting time perception and offer everyone deeper insights into their own experiences of the world. Our methodology, which combined theoretical judgments with advanced AI-powered interactions, led to a rich dataset on human time perception. By leveraging the power of ChatGPT-4 and its extensive training, we have gathered insights that push the boundaries of existing knowledge and pave the way for future research in this intriguing field.

4 Feature Descriptions

Analyzed Conditions: ADHD, Aging 18-29, Aging 30-49, Aging 50-69, Aging 70-89, Aging 90+, Alcohol, Alzheimer's Disease, Anxiety, Autism Type 1, Autism Type 2, Average, Bipolar Depressive, Bipolar Manic, Brain Damage, Brain Lesions, Caffeine, Chronic Pail disorder, Cocaine, Concentration Meditation, Depression (MDD), Dissociative Disorders, Elite Athlete, Emregency Event, Epilepsy, Fentanyl, Flow State, Graduate Student, Heroin, High IQ, Insomnia, Ketamione, Learrning Problems, Low IQ, LSD, Marijuana, MDMA (Ecstasy), Methamphetamine, Migraine, Morphine, MS, Musician, Nicotine, Oxycodone, Parkinson's Disease, Percription Antidepressants, Perscription Sleep Aids, Perscription Stimulants, Psilocybin, Psychosis, PTSD, Savant Type 1, Savant Type 2, Schizophrenia, Severe Cognitive Disability, Stress, TBI, Terminal, Tibetyan Meditation, Transcendental Meditation. Condition Features: Time Feels Faster or Slower (0 Average, 1 Slower, 2 Faster); Myelin Increase or Decrease (100% is Average); Speed (average speed associated with adequate myelination (75 meters per second); Speed with Myelination Increase or Decrease applied; Wavelength in meters = velocity divided by frequency; Number of Cycles = frequency times time; Total Distance (Delta waves) = wavelength times the number of cycles in kilometers; Delta Time (time in hours in an average day that a person with a given condition spends in that brainwave activity); Delta Frequency (the average frequency of delta waves); Items 4-9 repeat for theta waves, alpha waves, low beta waves, mid beta waves, high beta waves, and gamma waves; Total Time (24 hours); Cumulative Distance in kilometers; Calculated percent increase/decrease based on average person total distance; Rate = kilometers divided by hours; Cammie's Constant (altered perception of time less than 0 faster, greater than 0 slower); FeelsLike_Time; Delta_Distance; Delta_Time.

Feature names:

feature_names = 'Condition', 'Target', 'Myelin', 'Speed', 'Speed_M', 'D_WL', 'D_Cycle', 'D_Dist', 'D_Time', 'D_Freq', 'T_WL', 'T_Cycle', 'T_Dist', 'T_Time', 'T_Freq', 'A_WL', 'A_Cycle', 'A_Dist', 'A_Time', 'A_Freq', 'BL_WL', 'BL_Cycle', 'BL_Dist', 'BL_Time', 'BL_Freq', 'BM_WL', 'BM_Cycle', 'BM_Dist', 'BM_Time', 'BM_Freq', 'BH_WL', 'BH_Cycle', 'BH_Dist', 'BH_Time', 'BH_Freq', 'G_WL', 'G_Freq', 'G_Dist', 'G_Time', 'G_Freq.1', 'Total_Time', 'Total_Distance', ' Percent_Increase', 'Total_Rate', ' FeelsLike_Time', 'Delta_Time', 'Delta_Distance', 'Cammie_r'.

Cumulative Time and Initial Speed features are dropped becuase they are constants.

5 Machine Learning Approach

The utilization of Random Forest models in this research project stems from their inherent advantages in addressing the complexities of our multiclass classification problem. Random Forest models offer a robust and insightful approach to gaining a deeper understanding of the factors influencing classification accuracy. Their applicability was chosen based on the recognition that they provide a unique opportunity to unravel the importance of specific features, thereby informing subsequent feature selection and engineering efforts for the development of a deep learning neural network (NN).

Another compelling reason for embracing Random Forest models is their transparency in interpreting complex datasets. This transparency facilitates a clear comprehension of feature interactions and their contributions to overall accuracy. Beyond interpretability, these models serve as effective benchmarking tools, allowing for the establishment of performance baselines that aid in assessing the value of more intricate deep learning models. Moreover, their adeptness at handling class imbalances, often encountered in multiclass classification tasks, adds to their appeal, equipping us with the expertise needed to address real-world data scenarios effectively.

In summary, the choice to employ Random Forest models is a strategic one, driven by their capacity to provide critical insights into feature importance, enhance data interpretability, support model benchmarking, and tackle class imbalances. These models lay the groundwork for informing the design of a deep learning NN model capable of excelling in real-world scenarios with larger and more representative datasets.

6 Import packages and load data

[1]:
```
#general
import pandas as pd
import matplotlib
import matplotlib.pyplot as plt
import numpy as np
import random as rnd

# visualization
import seaborn as sns
import matplotlib.pyplot as plt
import graphviz
%matplotlib inline
sns.set()

# sklearn packages
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix
from sklearn.metrics import accuracy_score
from sklearn.metrics import classification_report
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import MinMaxScaler
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC, LinearSVC
from sklearn.ensemble import RandomForestClassifier, VotingClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.naive_bayes import GaussianNB
```

```
from sklearn.linear_model import Perceptron
from sklearn.linear_model import SGDClassifier
from sklearn.tree import DecisionTreeClassifier, export_graphviz
from sklearn.model_selection import GridSearchCV, cross_val_score
from sklearn.preprocessing import StandardScaler
from sklearn.ensemble import GradientBoostingClassifier

# imbalanced-learn
from imblearn.over_sampling import SMOTE

import warnings
warnings.filterwarnings("ignore")
```

[2]:
```
# Load the data
df = pd.read_csv("Temporal_MetricsI_CSV_Main_Datafile_12_2.csv")

df['Condition'] = df['Condition'].astype('str')

# Initial data exploration
print(df.head())
print(df.info())
print(df.describe())

# Check for missing values
print(df.isnull().sum())
```

```
     Condition  Target  Myelin  Speed  Speed_M  D_WL  D_Cycle  D_Dist  D_Time  \
0         ADHD       2    0.90     75    67.50  37.5    28800    1080     4.0
1  Aging 18-29       0    1.00     75    75.00  25.0    43200    1080     4.0
2  Aging 30-49       2    0.98     75    73.50  30.0    40500    1215     4.5
3  Aging 50-69       2    0.94     75    70.50  37.5    36000    1350     5.0
4  Aging 70-89       2    0.85     75    63.75  50.0    24300    1215     4.5

   D_Freq  …  G_Time  G_Freq.1  Total_Time  Total_Distance  \
0     2.0  …     1.0      40.0          24         6088.50
1     3.0  …     1.0      35.0          24         6480.00
2     2.5  …     0.5      35.0          24         6396.30
3     2.0  …     0.5      34.0          24         6237.00
4     1.5  …     0.0      33.0          24         5852.25

   Percent_Increase  Total_Rate  FeelsLike_Time  Delta_Time  Delta_Distance  \
0             -0.06      253.69           25.54       -1.45           -1.54
1              0.00      270.00           24.00        0.00            0.00
2             -0.01      266.51           24.31       -0.31           -0.31
3             -0.04      259.88           24.94       -0.90           -0.94
4             -0.10      243.84           26.57       -2.33           -2.57

   Cammie_r
```

```
0     -0.06
1      0.00
2     -0.01
3     -0.04
4     -0.10

[5 rows x 48 columns]
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 101 entries, 0 to 100
Data columns (total 48 columns):
 #   Column           Non-Null Count  Dtype
---  ------           --------------  -----
 0   Condition        101 non-null    object
 1   Target           101 non-null    int64
 2   Myelin           101 non-null    float64
 3   Speed            101 non-null    int64
 4   Speed_M          101 non-null    float64
 5   D_WL             101 non-null    float64
 6   D_Cycle          101 non-null    int64
 7   D_Dist           101 non-null    int64
 8   D_Time           101 non-null    float64
 9   D_Freq           101 non-null    float64
 10  T_WL             101 non-null    float64
 11  T_Cycle          101 non-null    int64
 12  T_Dist           101 non-null    int64
 13  T_Time           101 non-null    float64
 14  T_Freq           101 non-null    float64
 15  A_WL             101 non-null    float64
 16  A_Cycle          101 non-null    int64
 17  A_Dist           101 non-null    float64
 18  A_Time           101 non-null    float64
 19  A_Freq           101 non-null    float64
 20  BL_WL            101 non-null    float64
 21  BL_Cycle         101 non-null    int64
 22  BL_Dist          101 non-null    float64
 23  BL_Time          101 non-null    float64
 24  BL_Freq          101 non-null    float64
 25  BM_WL            101 non-null    float64
 26  BM_Cycle         101 non-null    int64
 27  BM_Dist          101 non-null    float64
 28  BM_Time          101 non-null    float64
 29  BM_Freq          101 non-null    float64
 30  BH_WL            101 non-null    float64
 31  BH_Cycle         101 non-null    int64
 32  BH_Dist          101 non-null    float64
 33  BH_Time          101 non-null    float64
 34  BH_Freq          101 non-null    float64
 35  G_WL             101 non-null    float64
```

```
 36  G_Freq            101 non-null    int64
 37  G_Dist            101 non-null    float64
 38  G_Time            101 non-null    float64
 39  G_Freq.1          101 non-null    float64
 40  Total_Time        101 non-null    int64
 41  Total_Distance    101 non-null    float64
 42  Percent_Increase  101 non-null    float64
 43  Total_Rate        101 non-null    float64
 44  FeelsLike_Time    101 non-null    float64
 45  Delta_Time        101 non-null    float64
 46  Delta_Distance    101 non-null    float64
 47  Cammie_r          101 non-null    float64
dtypes: float64(35), int64(12), object(1)
memory usage: 38.0+ KB
None
           Target     Myelin       Speed     Speed_M        D_WL  \
count  101.000000  101.00000  101.000000  101.000000  101.000000
mean     1.534653    0.95198   75.009901   71.353168   38.366337
std      0.625529    0.11562    0.099504    8.658401   12.730061
min      0.000000    0.50000   75.000000   37.500000   25.000000
25%      1.000000    0.90000   75.000000   67.500000   30.000000
50%      2.000000    0.97000   75.000000   72.750000   37.500000
75%      2.000000    1.03000   75.000000   76.880000   50.000000
max      2.000000    1.25000   76.000000   93.750000   75.000000

             D_Cycle       D_Dist      D_Time      D_Freq        T_WL  …  \
count     101.000000   101.000000  101.000000  101.000000  101.000000  …
mean    33558.415842  1185.594059    4.391089    2.138614   14.873861  …
std     13965.573868   335.932499    1.244194    0.596317    2.417425  …
min      5400.000000   405.000000    1.500000    1.000000   10.710000  …
25%     27000.000000  1080.000000    4.000000    1.500000   12.500000  …
50%     28800.000000  1080.000000    4.000000    2.000000   15.000000  …
75%     37800.000000  1350.000000    5.000000    2.500000   16.670000  …
max    108000.000000  2700.000000   10.000000    3.000000   18.750000  …

           G_Time    G_Freq.1  Total_Time  Total_Distance  Percent_Increase  \
count  101.000000  101.000000       101.0      101.000000        101.000000
mean     1.128713   36.004950        24.0     6283.951980         -0.030594
std      0.832627    3.707759         0.0      458.560731          0.071468
min      0.000000   30.000000        24.0     4860.000000         -0.250000
25%      0.500000   33.000000        24.0     6048.000000         -0.070000
50%      1.000000   35.500000        24.0     6322.050000         -0.020000
75%      2.000000   40.000000        24.0     6588.000000          0.020000
max      4.000000   45.000000        24.0     7492.500000          0.160000

       Total_Rate  FeelsLike_Time  Delta_Time  Delta_Distance    Cammie_r
count  101.000000      101.000000   101.00000      101.000000  101.000000
mean   261.831980       24.876040    -0.72604       -0.886040   -0.030594
```

```
std     19.106928        1.918818     1.69900         1.917176    0.071468
min    202.500000       20.760000    -6.00000        -8.000000   -0.250000
25%    252.000000       23.610000    -1.60000        -1.710000   -0.070000
50%    263.420000       24.460000    -0.59000        -0.600000   -0.020000
75%    274.500000       25.710000     0.40000         0.390000    0.020000
max    312.190000       32.000000     3.75000         3.240000    0.160000

[8 rows x 47 columns]
Condition            0
Target               0
Myelin               0
Speed                0
Speed_M              0
D_WL                 0
D_Cycle              0
D_Dist               0
D_Time               0
D_Freq               0
T_WL                 0
T_Cycle              0
T_Dist               0
T_Time               0
T_Freq               0
A_WL                 0
A_Cycle              0
A_Dist               0
A_Time               0
A_Freq               0
BL_WL                0
BL_Cycle             0
BL_Dist              0
BL_Time              0
BL_Freq              0
BM_WL                0
BM_Cycle             0
BM_Dist              0
BM_Time              0
BM_Freq              0
BH_WL                0
BH_Cycle             0
BH_Dist              0
BH_Time              0
BH_Freq              0
G_WL                 0
G_Freq               0
G_Dist               0
G_Time               0
G_Freq.1             0
```

```
Total_Time          0
Total_Distance      0
Percent_Increase    0
Total_Rate          0
FeelsLike_Time      0
Delta_Time          0
Delta_Distance      0
Cammie_r            0
dtype: int64
```

[3]:
```python
# Convert column names to a list
column_names = df.columns.tolist()

# Print the list of column names
print(column_names)
```

```
['Condition', 'Target', 'Myelin', 'Speed', 'Speed_M', 'D_WL', 'D_Cycle',
'D_Dist', 'D_Time', 'D_Freq', 'T_WL', 'T_Cycle', 'T_Dist', 'T_Time', 'T_Freq',
'A_WL', 'A_Cycle', 'A_Dist', 'A_Time', 'A_Freq', 'BL_WL', 'BL_Cycle', 'BL_Dist',
'BL_Time', 'BL_Freq', 'BM_WL', 'BM_Cycle', 'BM_Dist', 'BM_Time', 'BM_Freq',
'BH_WL', 'BH_Cycle', 'BH_Dist', 'BH_Time', 'BH_Freq', 'G_WL', 'G_Freq',
'G_Dist', 'G_Time', 'G_Freq.1', 'Total_Time', 'Total_Distance',
'Percent_Increase', 'Total_Rate', 'FeelsLike_Time', 'Delta_Time',
'Delta_Distance', 'Cammie_r']
```

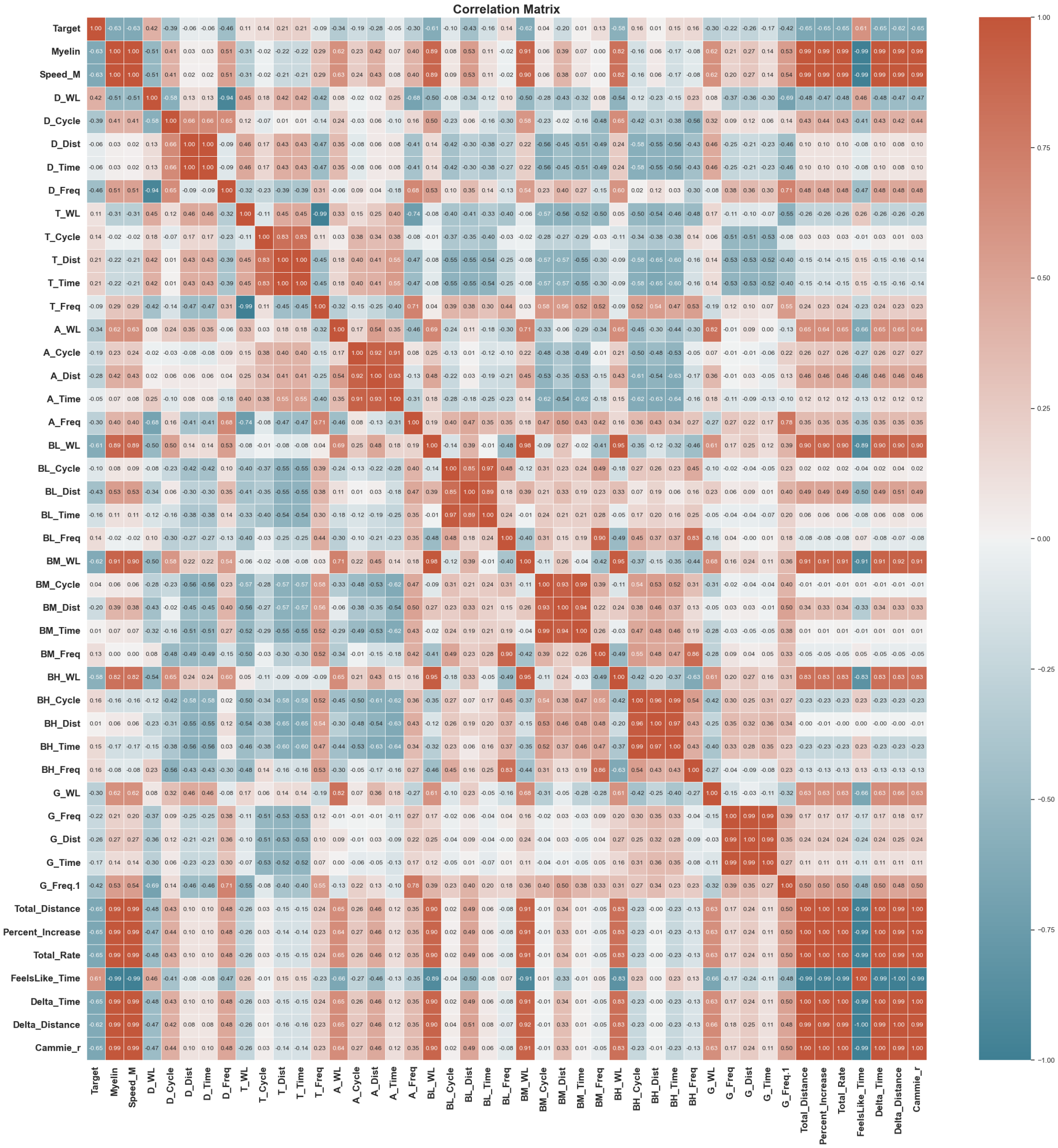
[4]:
```python
df.drop(['Speed', 'Total_Time'], axis=1) #these are constants

# Convert column names to a list
column_names = df.columns.tolist()

# Print the list of column names
print(column_names)
```

```
['Condition', 'Target', 'Myelin', 'Speed', 'Speed_M', 'D_WL', 'D_Cycle',
'D_Dist', 'D_Time', 'D_Freq', 'T_WL', 'T_Cycle', 'T_Dist', 'T_Time', 'T_Freq',
'A_WL', 'A_Cycle', 'A_Dist', 'A_Time', 'A_Freq', 'BL_WL', 'BL_Cycle', 'BL_Dist',
'BL_Time', 'BL_Freq', 'BM_WL', 'BM_Cycle', 'BM_Dist', 'BM_Time', 'BM_Freq',
'BH_WL', 'BH_Cycle', 'BH_Dist', 'BH_Time', 'BH_Freq', 'G_WL', 'G_Freq',
'G_Dist', 'G_Time', 'G_Freq.1', 'Total_Time', 'Total_Distance',
'Percent_Increase', 'Total_Rate', 'FeelsLike_Time', 'Delta_Time',
'Delta_Distance', 'Cammie_r']
```

7 Exploratory Data Analysis

[5]:
```python
# Correlation matrix
corr_matrix = df[['Condition', 'Target', 'Myelin', 'Speed_M', 'D_WL',
                  'D_Cycle', 'D_Dist', 'D_Time', 'D_Freq', 'T_WL', 'T_Cycle',
                  'T_Dist', 'T_Time', 'T_Freq', 'A_WL', 'A_Cycle', 'A_Dist',
```

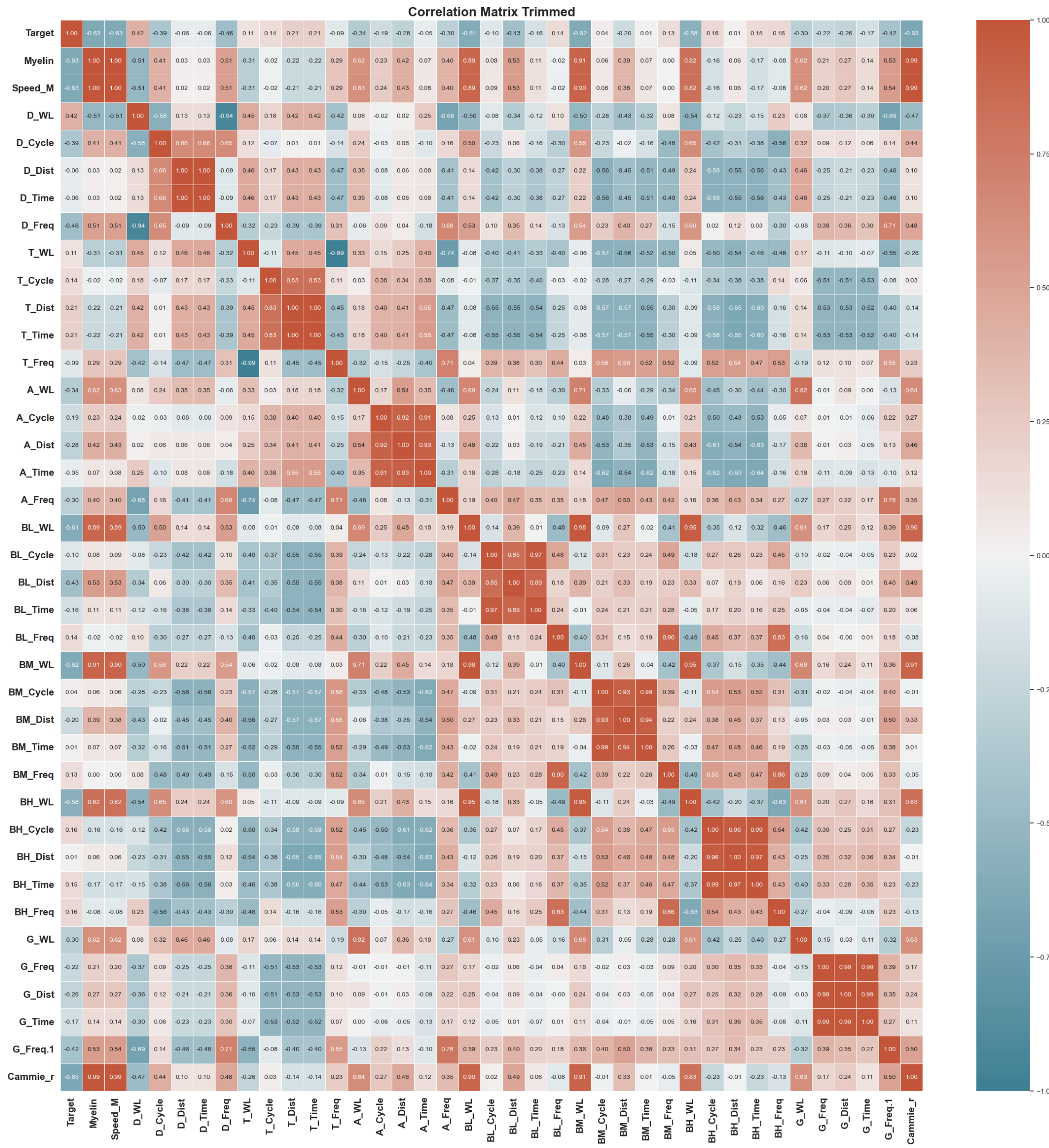
```
                  'A_Time', 'A_Freq', 'BL_WL', 'BL_Cycle', 'BL_Dist', 'BL_Time',
                  'BL_Freq', 'BM_WL', 'BM_Cycle', 'BM_Dist', 'BM_Time',␣
 ↪'BM_Freq',
                  'BH_WL', 'BH_Cycle', 'BH_Dist', 'BH_Time', 'BH_Freq', 'G_WL',
                  'G_Freq', 'G_Dist', 'G_Time', 'G_Freq.1', 'Total_Distance',
                  'Percent_Increase', 'Total_Rate', 'FeelsLike_Time',␣
 ↪'Delta_Time',
                  'Delta_Distance', 'Cammie_r'

]].corr(numeric_only=True)

# Plotting
plt.figure(figsize=(30,30))
sns.heatmap(
    corr_matrix,
    annot=True,
    fmt=".2f",
    cmap=sns.diverging_palette(220, 20, as_cmap=True),
    vmin=-1,
    vmax=1,
    annot_kws={"size": 10},  # Adjust font size and weight for the annotations
    linewidths=0.5  # Adjusts line width for gridlines
)
plt.xticks(fontsize=15, weight='bold')  # Adjust x-tick label font size and␣
 ↪weight
plt.yticks(fontsize=15, weight='bold')  # Adjust y-tick label font size and␣
 ↪weight
plt.title('Correlation Matrix', fontsize=20, weight="bold")
plt.show()
```



[6]:

```
# Correlation matrix
corr_matrix_trimmed = df[['Condition', 'Target', 'Myelin', 'Speed_M', 'D_WL',
                    'D_Cycle', 'D_Dist', 'D_Time', 'D_Freq', 'T_WL', 'T_Cycle',
                    'T_Dist', 'T_Time', 'T_Freq', 'A_WL', 'A_Cycle', 'A_Dist',
                    'A_Time', 'A_Freq', 'BL_WL', 'BL_Cycle', 'BL_Dist', 'BL_Time',
                    'BL_Freq', 'BM_WL', 'BM_Cycle', 'BM_Dist', 'BM_Time',␣
 ↪'BM_Freq',
                    'BH_WL', 'BH_Cycle', 'BH_Dist', 'BH_Time', 'BH_Freq', 'G_WL',
                    'G_Freq', 'G_Dist', 'G_Time', 'G_Freq.1', 'Cammie_r'
```

```
]].corr(numeric_only=True)

# Plotting
plt.figure(figsize=(30,30))
sns.heatmap(
    corr_matrix_trimmed,
    annot=True,
    fmt=".2f",
    cmap=sns.diverging_palette(220, 20, as_cmap=True),
    vmin=-1,
    vmax=1,
    annot_kws={"size": 10},  # Adjust font size and weight for the annotations
    linewidths=0.5  # Adjusts line width for gridlines
)
plt.xticks(fontsize=15, weight='bold')  # Adjust x-tick label font size and␣
↪weight
plt.yticks(fontsize=15, weight='bold')  # Adjust y-tick label font size and␣
↪weight
plt.title('Correlation Matrix Trimmed', fontsize=20, weight="bold")
plt.show()
```

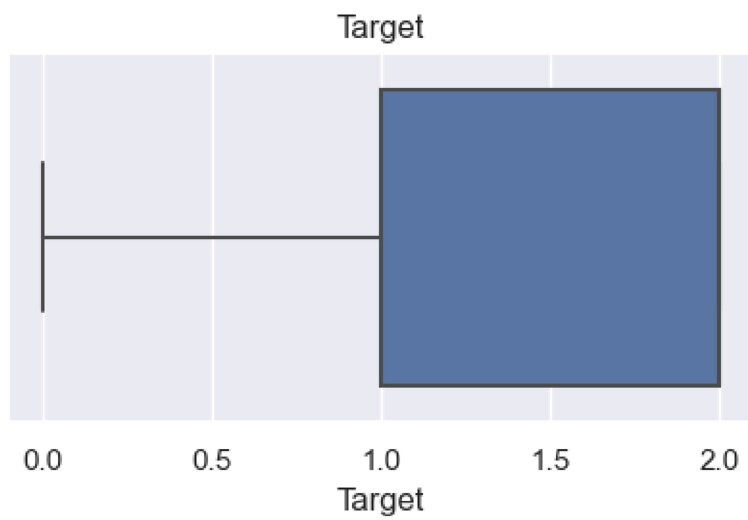


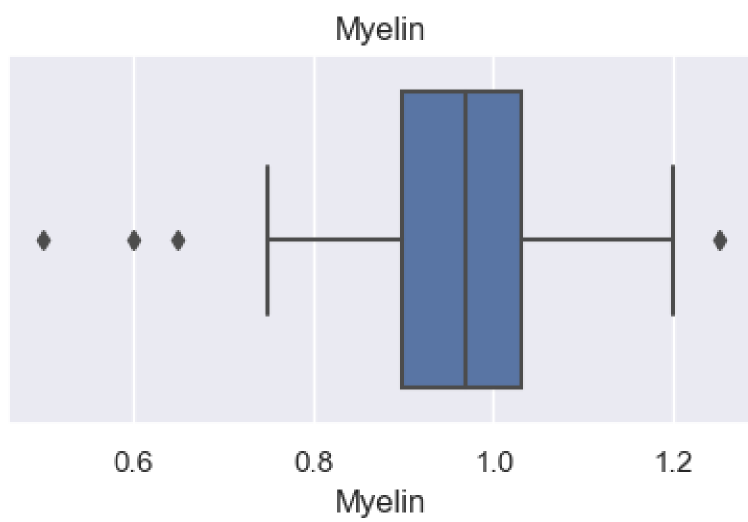
[7]:
```
# Visualization
# Histogram for all features
df[['Condition', 'Target', 'Myelin', 'Speed_M', 'D_WL',
                   'D_Cycle', 'D_Dist', 'D_Time', 'D_Freq', 'T_WL', 'T_Cycle',
                   'T_Dist', 'T_Time', 'T_Freq', 'A_WL', 'A_Cycle', 'A_Dist',
                   'A_Time', 'A_Freq', 'BL_WL', 'BL_Cycle', 'BL_Dist', 'BL_Time',
                   'BL_Freq', 'BM_WL', 'BM_Cycle', 'BM_Dist', 'BM_Time',␣
 ↪'BM_Freq',
                   'BH_WL', 'BH_Cycle', 'BH_Dist', 'BH_Time', 'BH_Freq', 'G_WL',
```

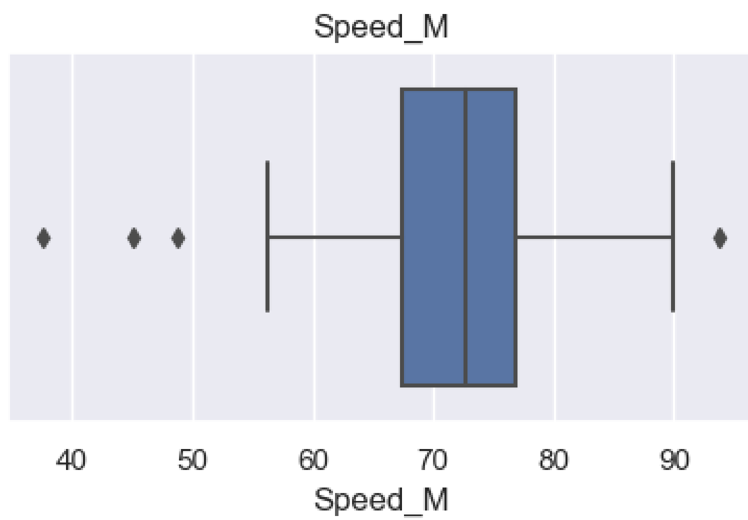
```
                  'G_Freq', 'G_Dist', 'G_Time', 'G_Freq.1', 'Cammie_r']].
↪hist(figsize=(20,20))
plt.tight_layout()  # Ensures that plots don't overlap
plt.show()
```

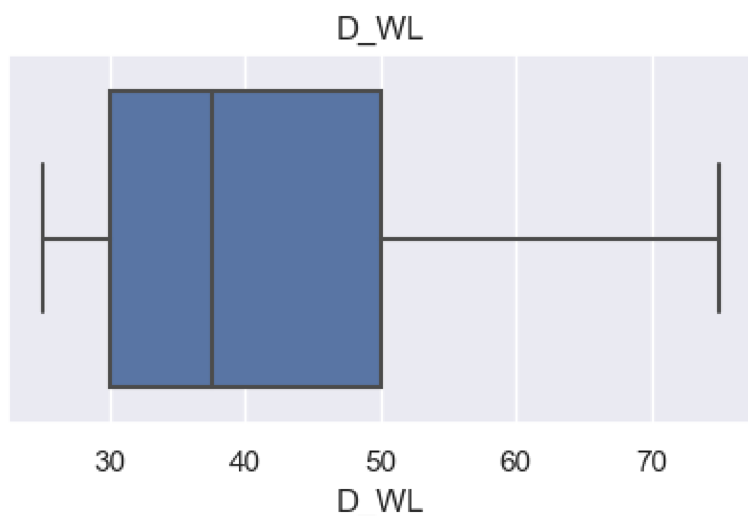
[8]:
```
# Boxplot for all features to identify outliers
for column in df[['Condition', 'Target', 'Myelin', 'Speed_M', 'D_WL',
                  'D_Cycle', 'D_Dist', 'D_Time', 'D_Freq', 'T_WL', 'T_Cycle',
                  'T_Dist', 'T_Time', 'T_Freq', 'A_WL', 'A_Cycle', 'A_Dist',
                  'A_Time', 'A_Freq', 'BL_WL', 'BL_Cycle', 'BL_Dist', 'BL_Time',
                  'BL_Freq', 'BM_WL', 'BM_Cycle', 'BM_Dist', 'BM_Time',␣
↪'BM_Freq',
```

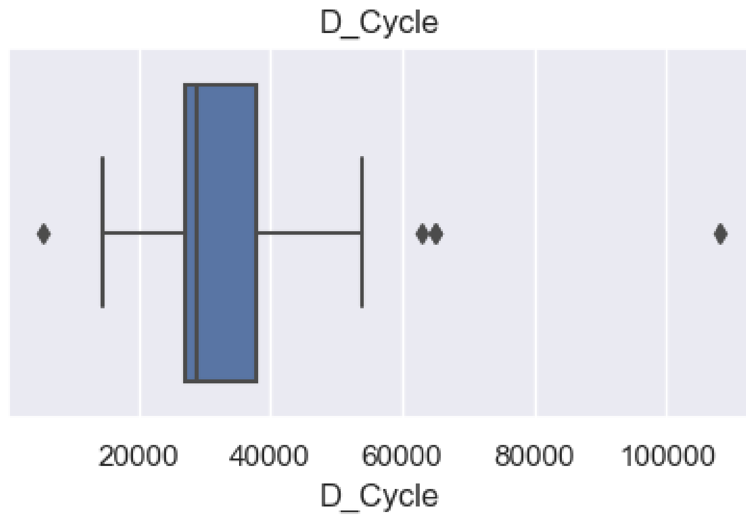
```
                 'BH_WL', 'BH_Cycle', 'BH_Dist', 'BH_Time', 'BH_Freq', 'G_WL',
                 'G_Freq', 'G_Dist', 'G_Time', 'G_Freq.1', 'Cammie_r']].
↪columns:
   if df[column].dtype in ['float64', 'int64']:  # only plot for numeric␣
↪columns
       plt.figure(figsize=(5, 2.5))
       sns.boxplot(x=column, data=df)
       plt.title(column)
       plt.show()
```

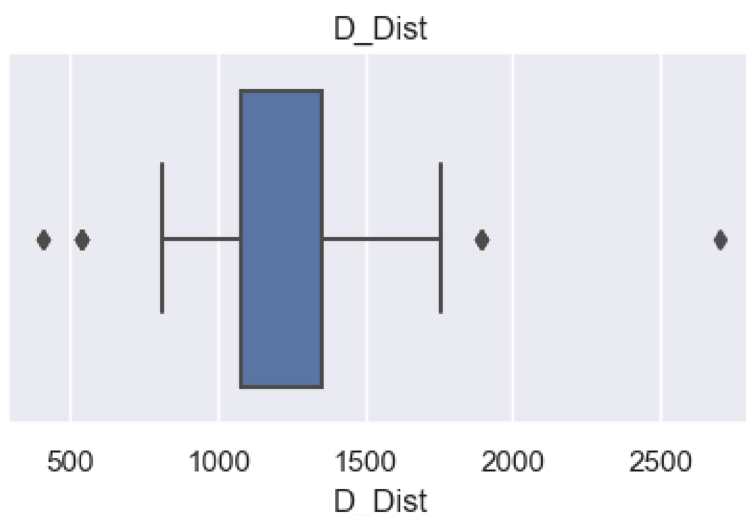


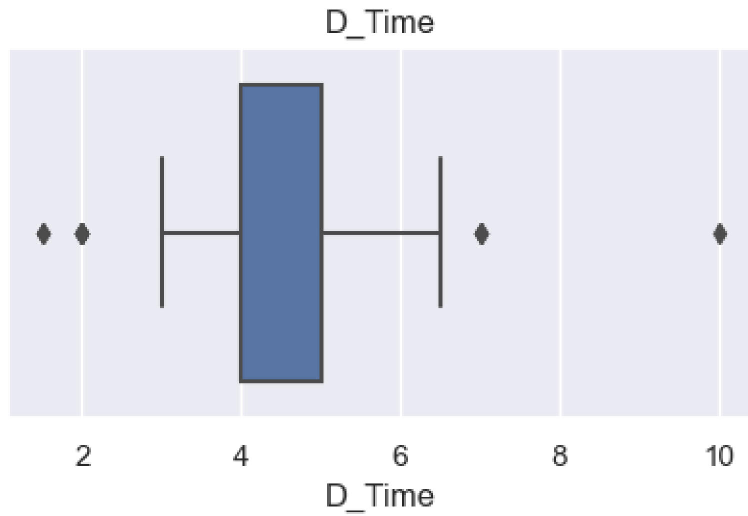


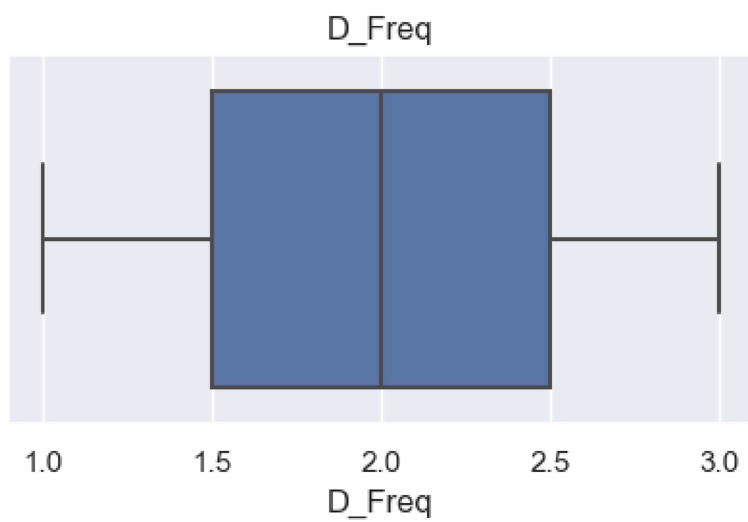

Speed_M
40
50
60
70
80
90
Speed_M

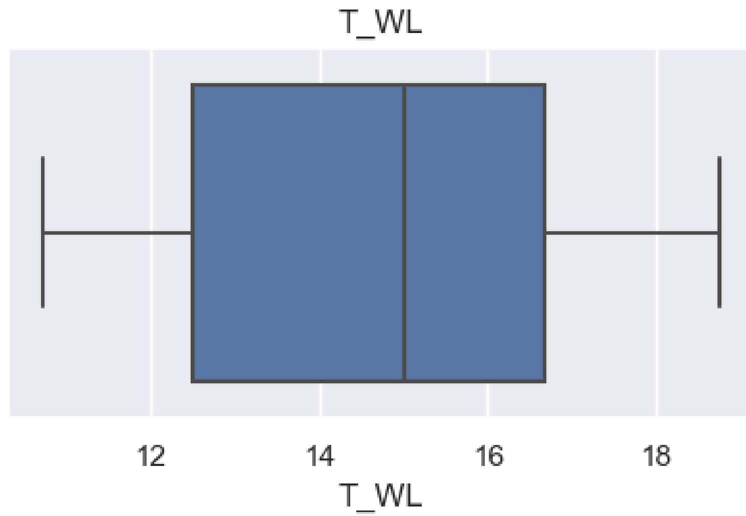

D_WL
30
40
50
60
70
D_WL

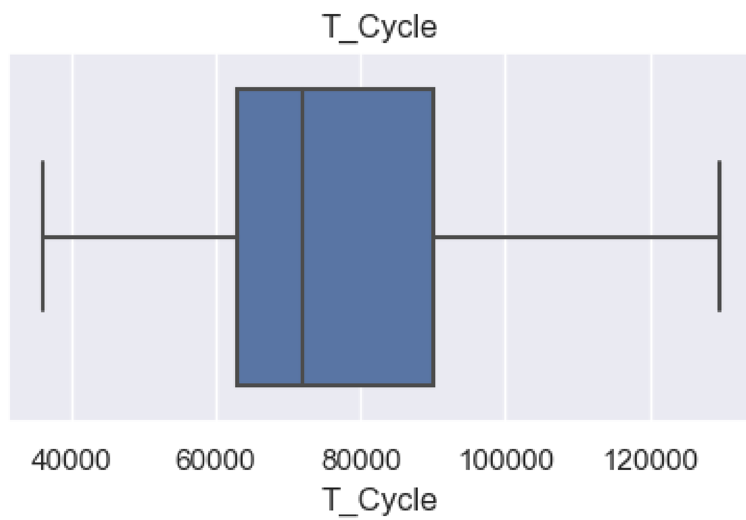

D_Cycle
20000
40000
60000
80000
100000
D_Cycle

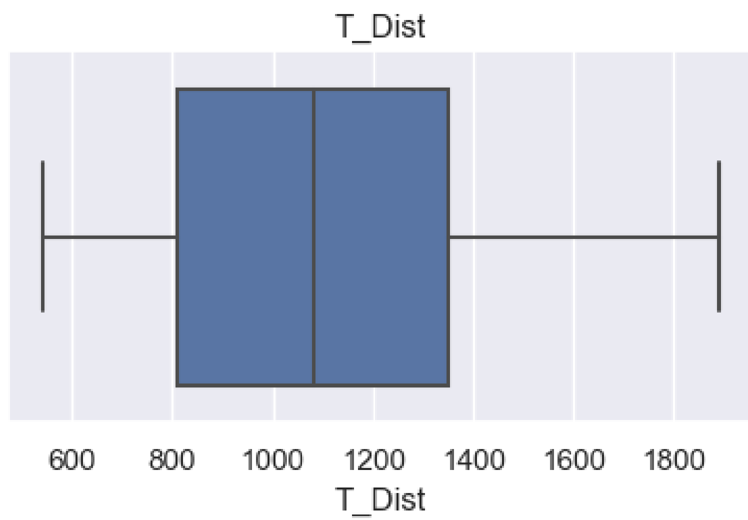

D_Dist
500
1000
1500
2000
2500
D_Dist

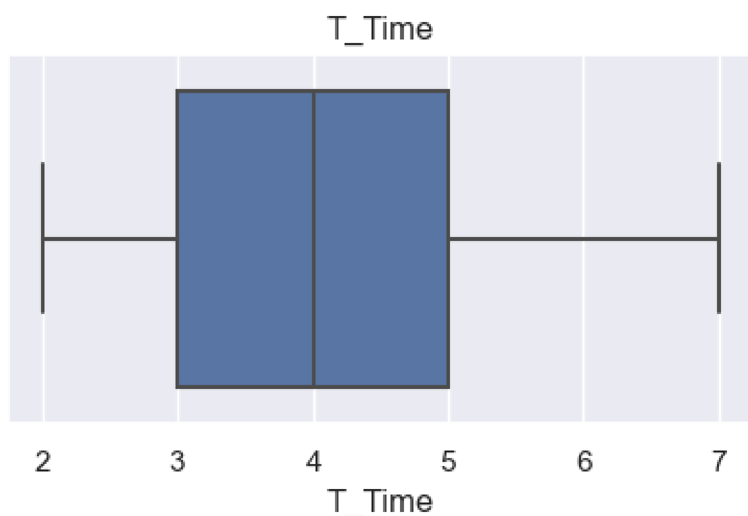

D_Time
2
4
6
8
10
D_Time

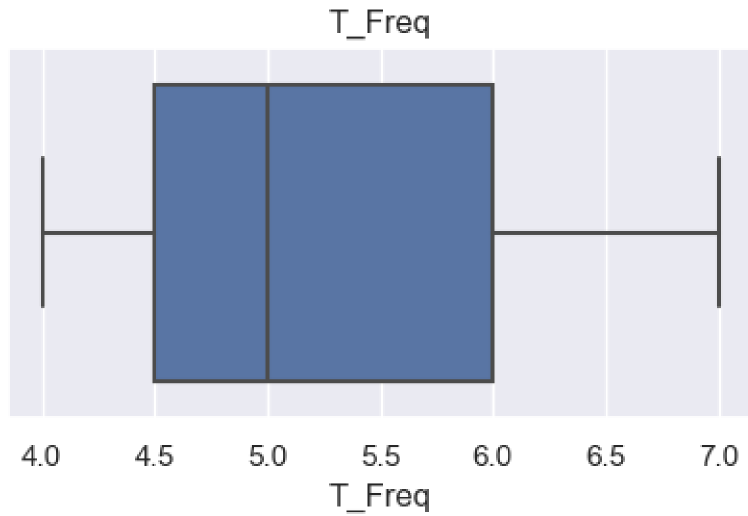

D_Freq
1.0
1.5
2.0
2.5
3.0
D_Freq

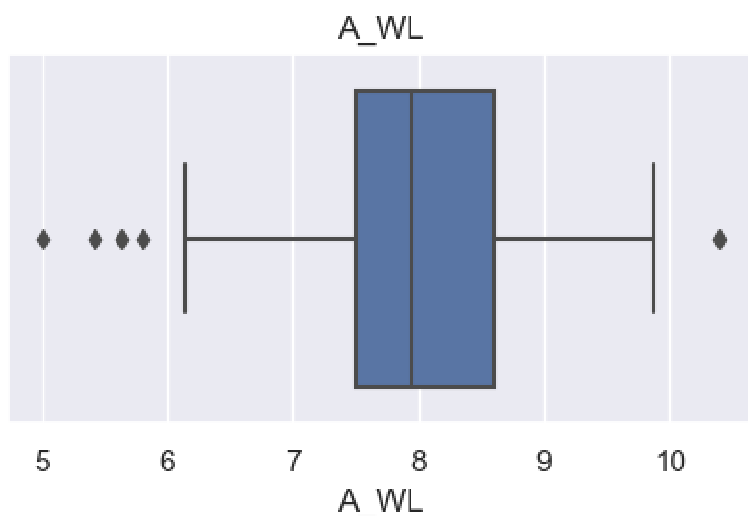

T_WL
12
14
16
18
T_WL

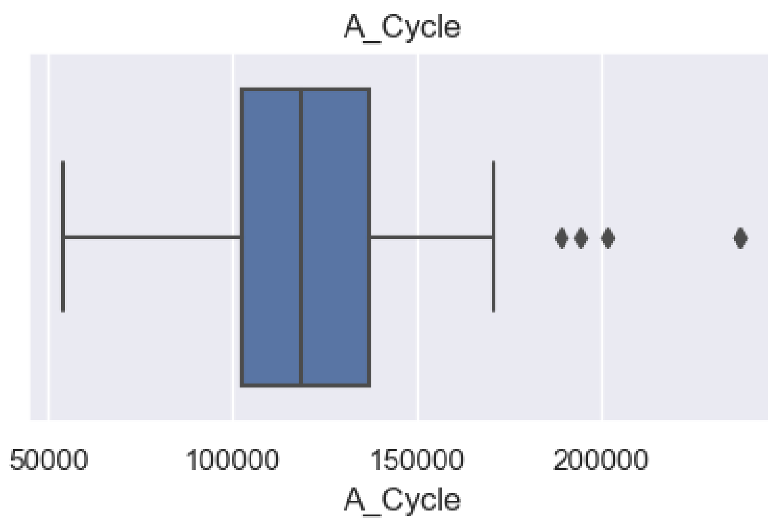

T_Cycle
40000
60000
80000
100000
120000
T_Cycle

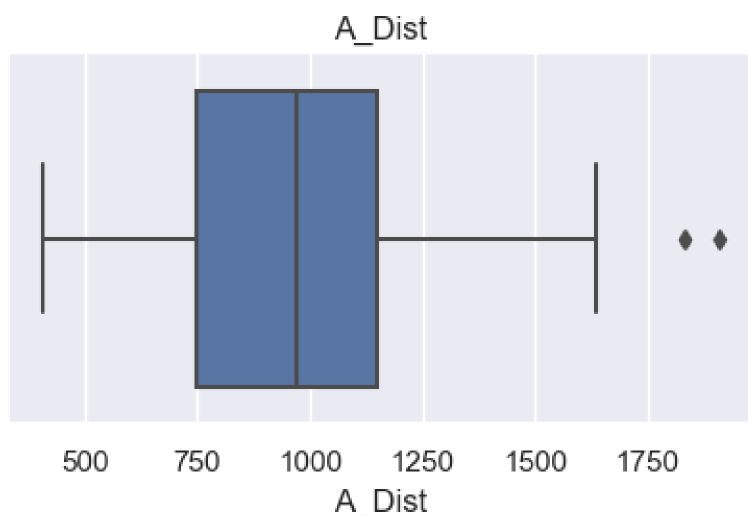

T_Dist
600
800
1000
1200
1400
1600
1800
T_Dist

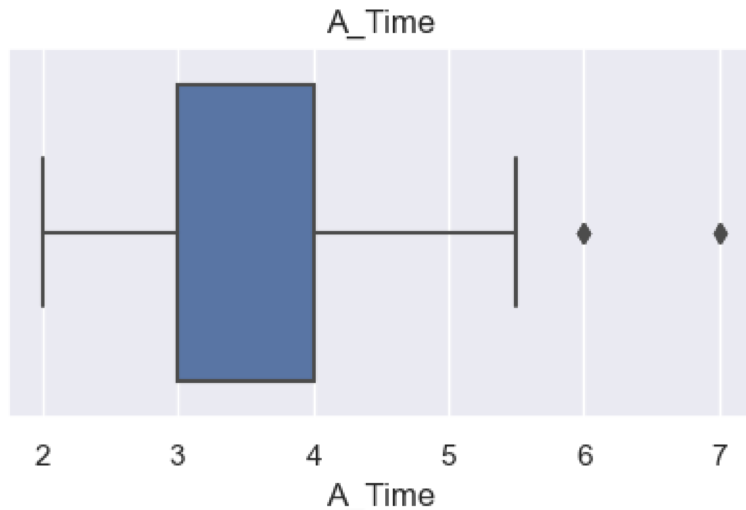

T_Time
2
3
4
5
6
7
T_Time

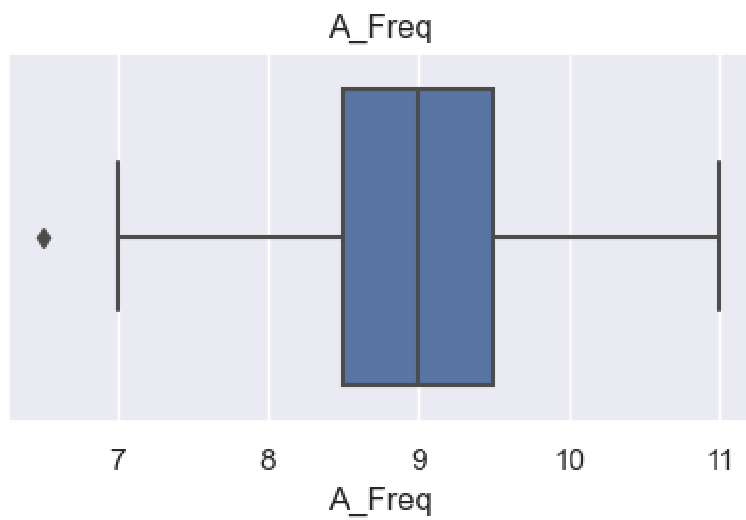

T_Freq
4.0
4.5
5.0
5.5
6.0
6.5
7.0
T_Freq

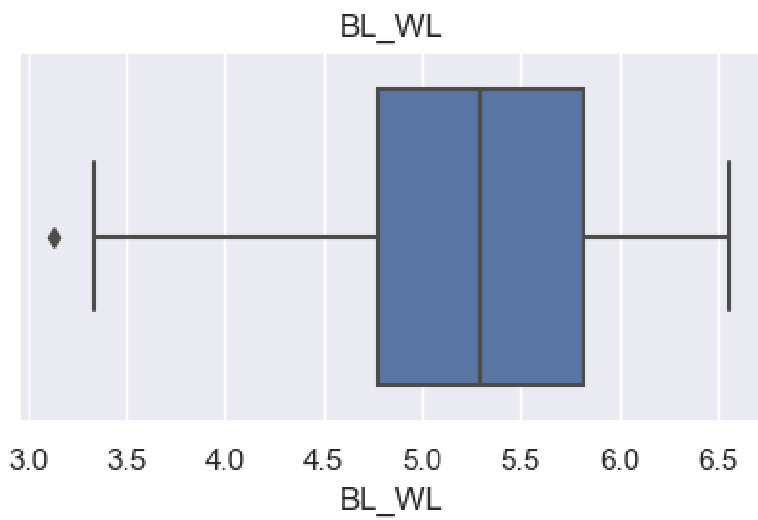

A_WL
5
6
7
8
9
10
A_WL

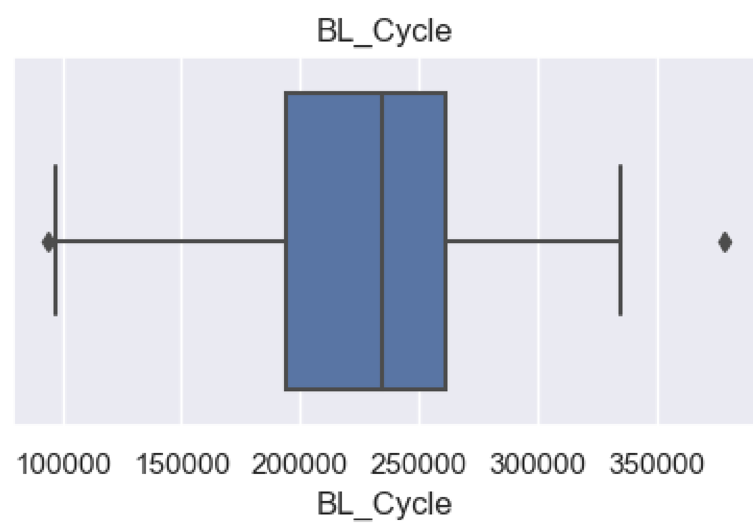

A_Cycle
50000
100000
150000
200000
A_Cycle

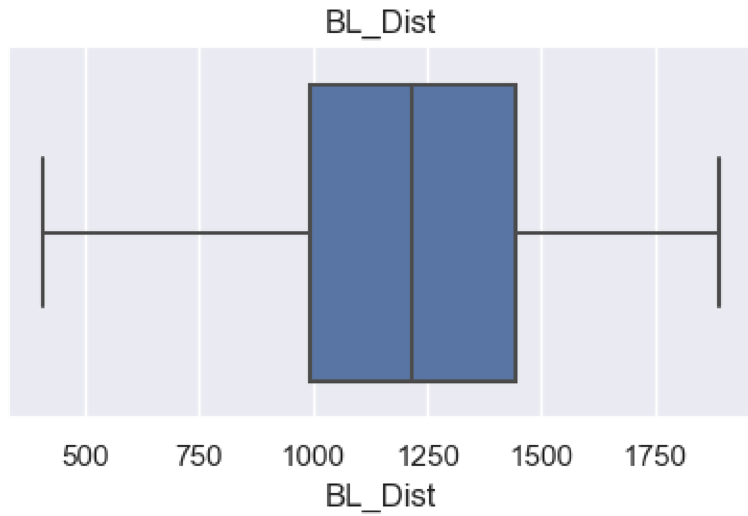

A_Dist
500
750
1000
1250
1500
1750
A_Dist

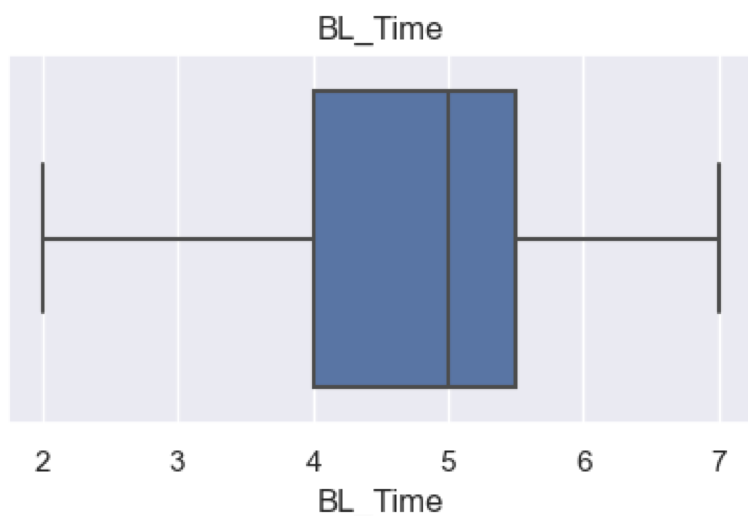

A_Time
2
3
4
5
6
7
A_Time

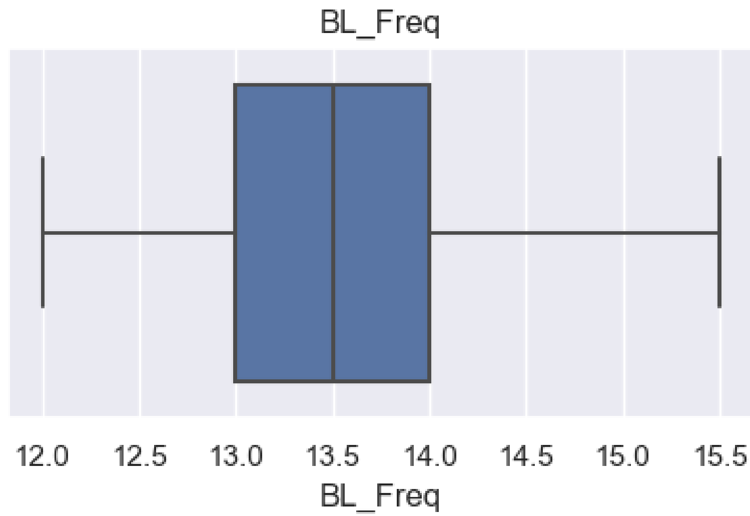

A_Freq
7
8
9
10
11
A_Freq

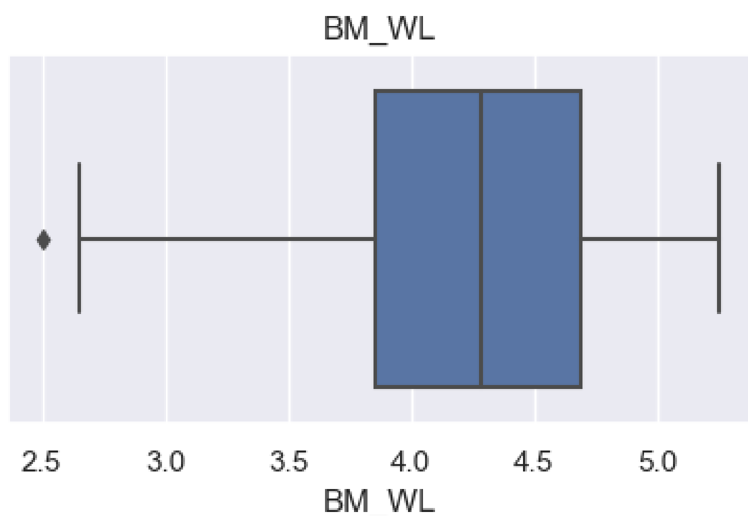

BL_WL
3.0
3.5
4.0
4.5
5.0
5.5
6.0
6.5
BL_WL

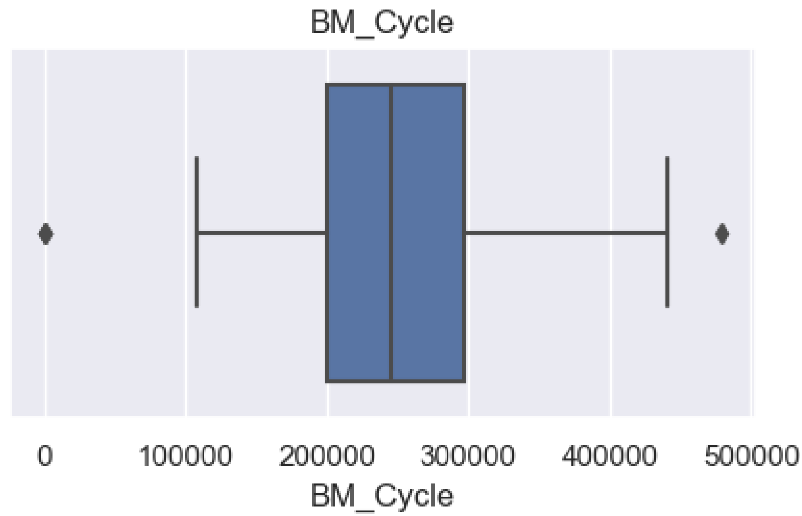

BL_Cycle
100000
150000
200000
250000
300000
350000
BL_Cycle

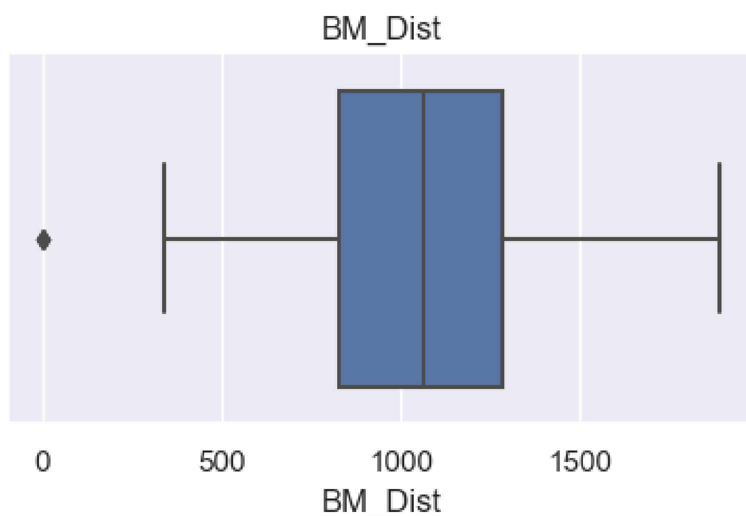

BL_Dist
500
750
1000
1250
1500
1750
BL_Dist

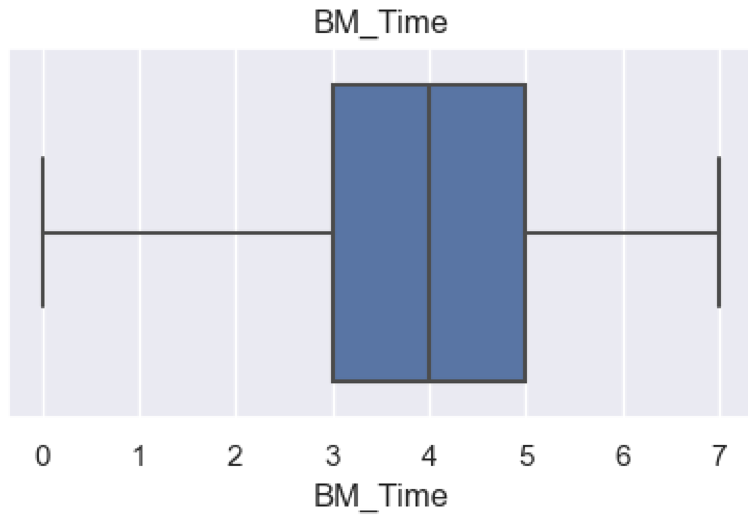

BL_Time
2
3
4
5
6
7
BL_Time

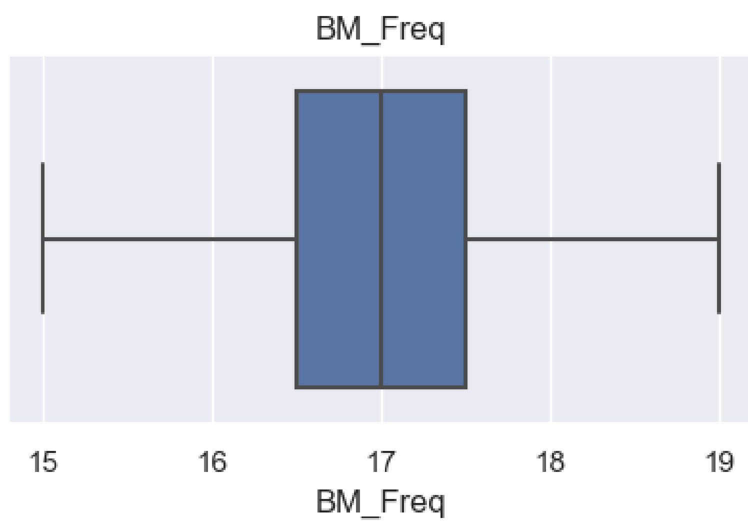

BL_Freq
12.0
12.5
13.0
13.5
14.0
14.5
15.0
15.5
BL_Freq

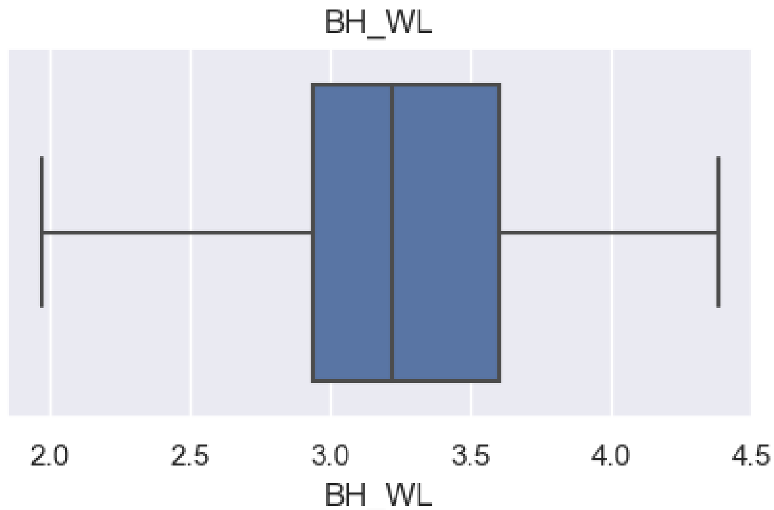

BM_WL
2.5
3.0
3.5
4.0
4.5
5.0
BM_WL

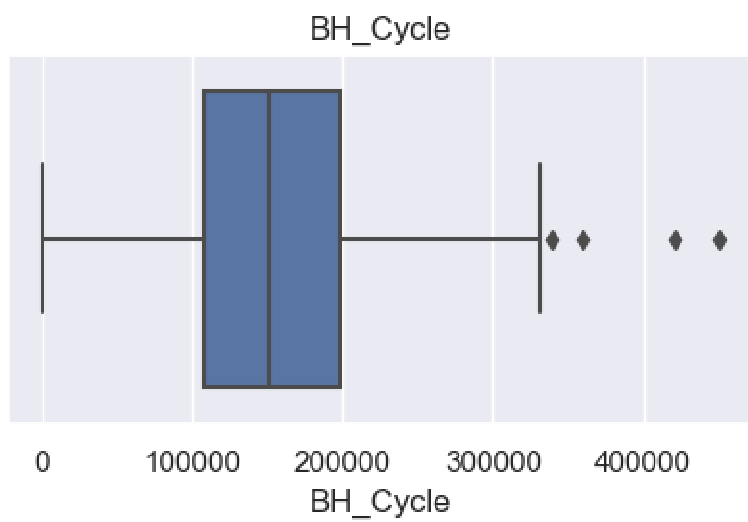

BM_Cycle
0
100000
200000
300000
400000
500000
BM_Cycle

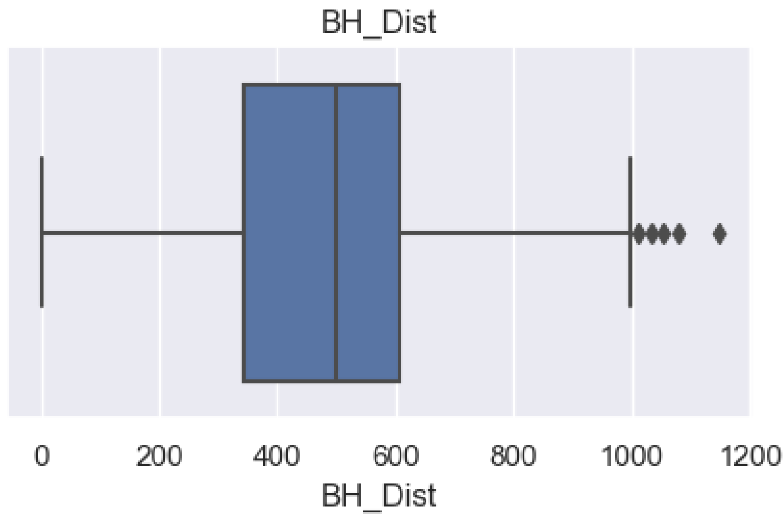

BM_Dist
0
500
1000
1500
BM_Dist

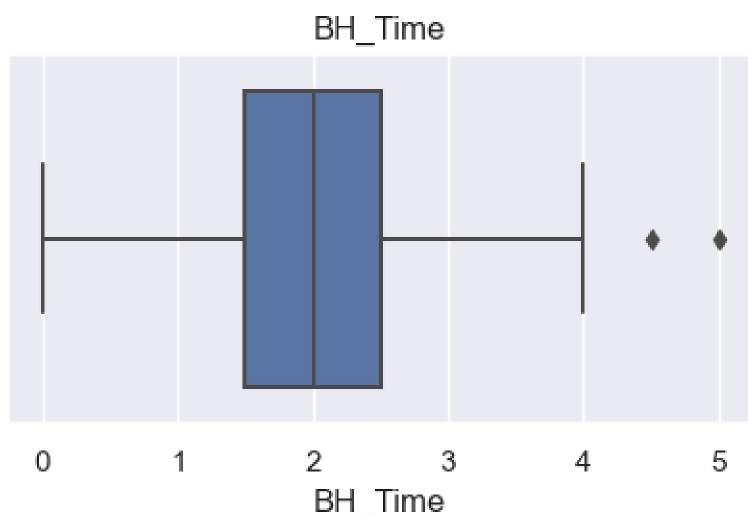

BM_Time
0
1
2
3
4
5
6
7
BM_Time

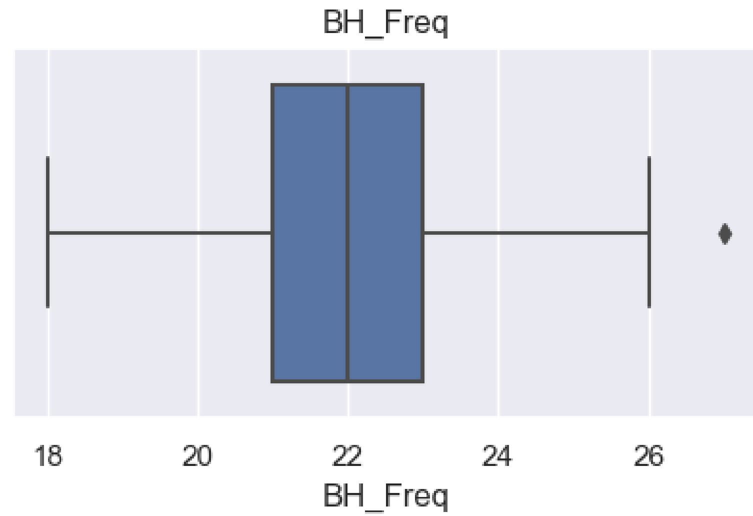

BM_Freq
15
16
17
18
19
BM_Freq

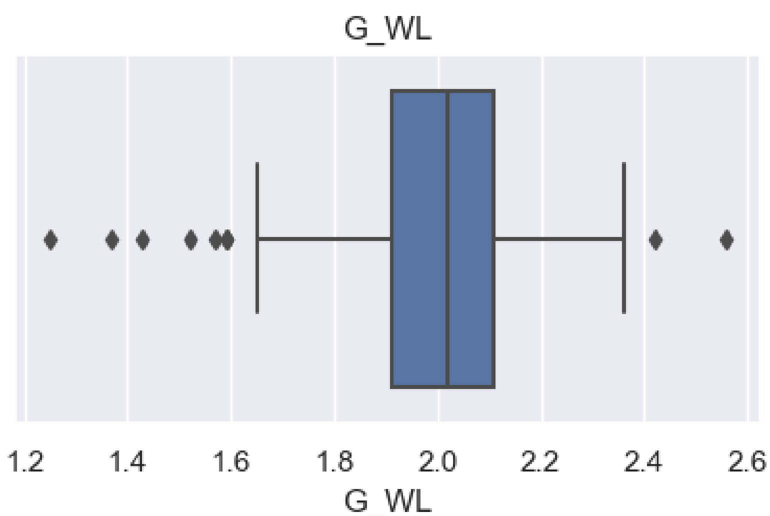

BH_WL
2.0
2.5
3.0
3.5
4.0
4.5
BH_WL

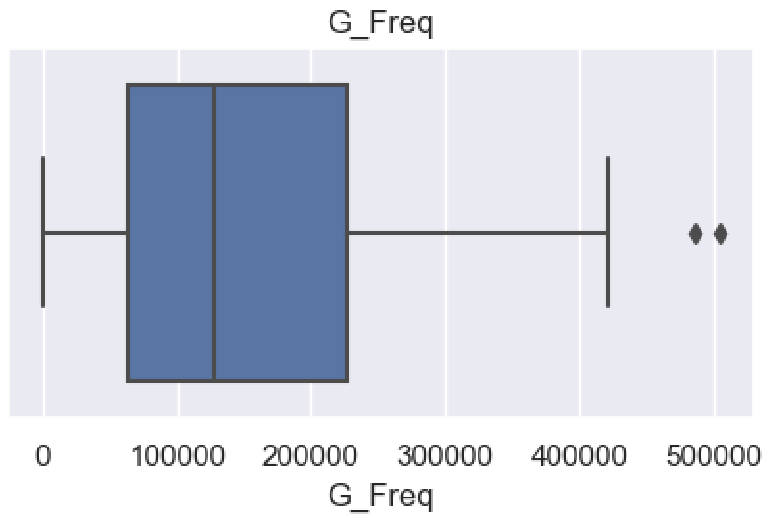

BH_Cycle
0
100000
200000
300000
400000
BH_Cycle

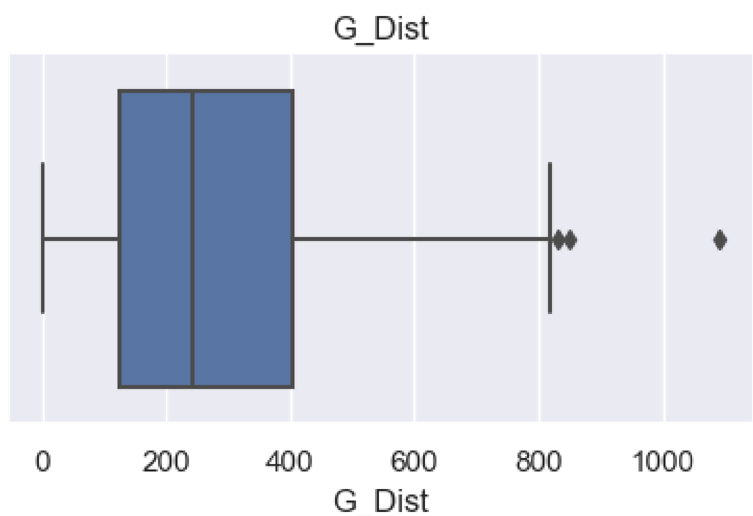

BH_Dist
0
200
400
600
800
1000
1200
BH_Dist

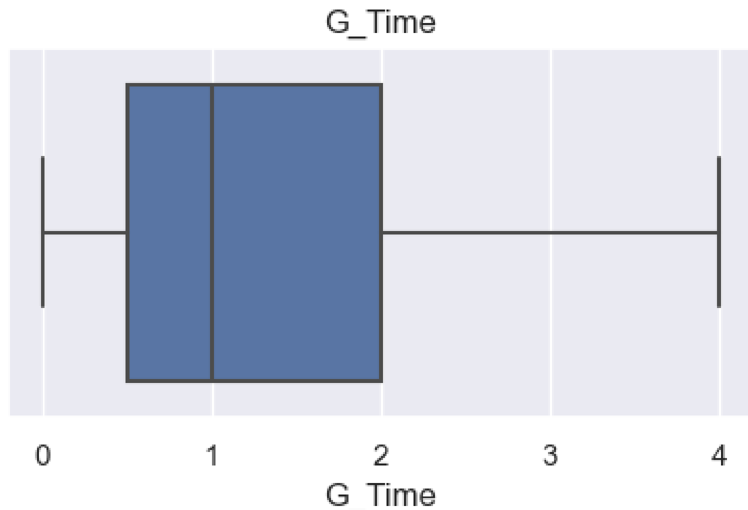

BH_Time
0
1
2
3
4
5
BH_Time

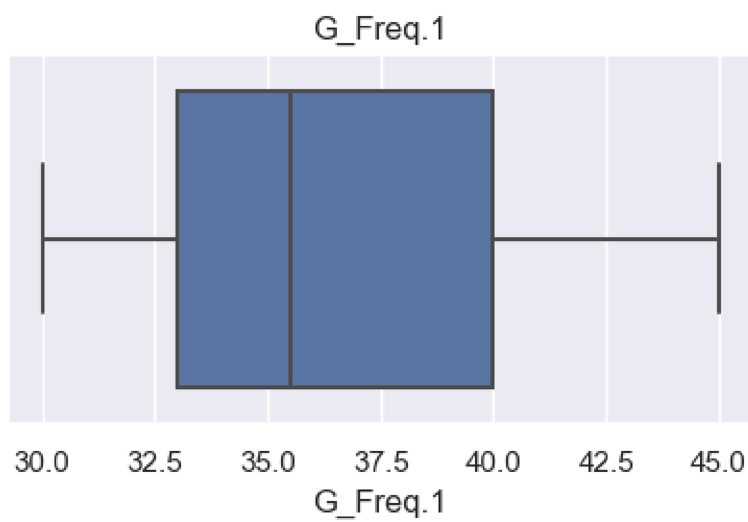

BH_Freq
18
20
22
24
26
BH_Freq

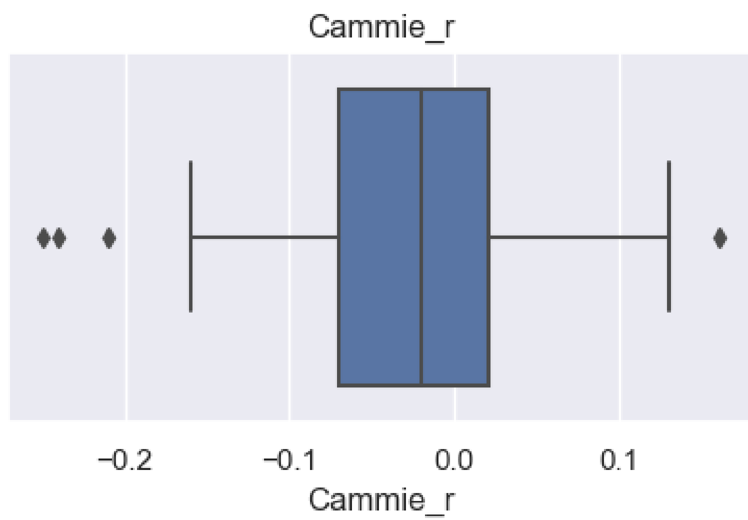

G_WL
1.2
1.4
1.6
1.8
2.0
2.2
2.4
2.6
G_WL


G_Freq
0
100000
200000
300000
400000
500000
G_Freq

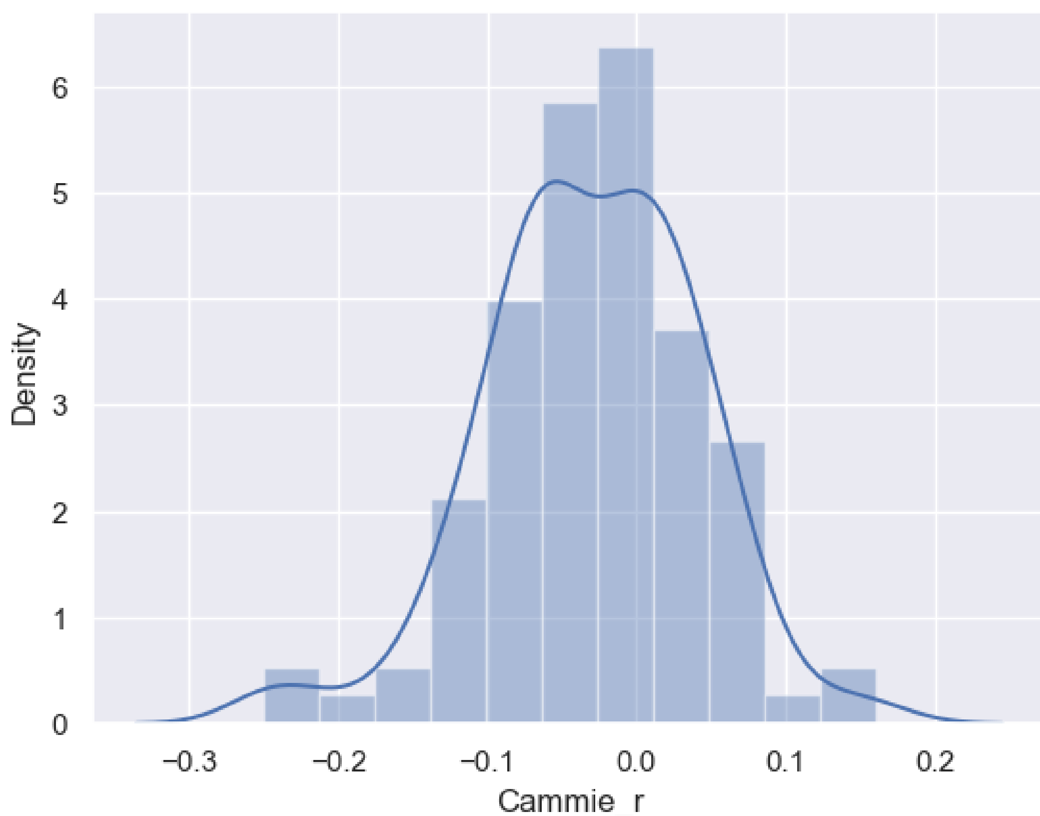

G_Dist
0
200
400
600
800
1000
G_Dist


G_Time
0
1
2
3
4
G_Time


G_Freq.1
30.0
32.5
35.0
37.5
40.0
42.5
45.0
G_Freq.1

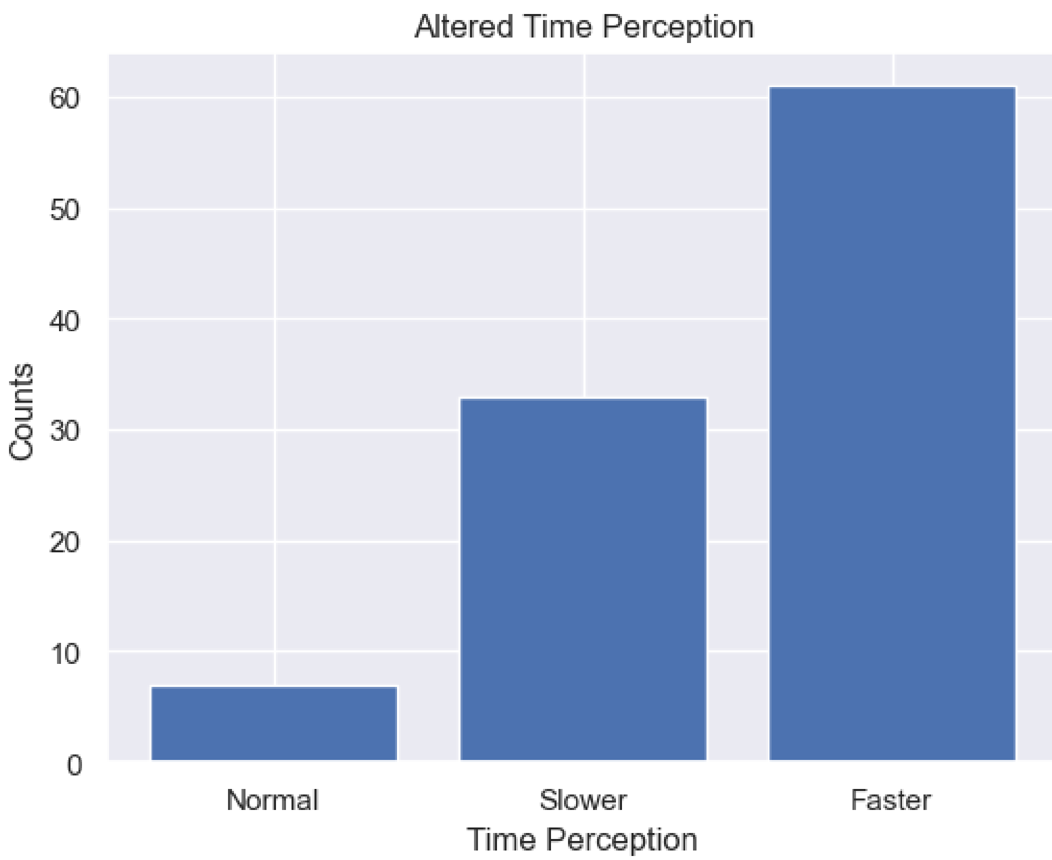


[9]:
```
# Distribution of time change constant
_ = sns.distplot(df.Cammie_r)
```



```
[10]: # Select features and define a subset
selected_columns = ['Target', 'D_Dist', 'D_Time',
                    'T_Dist', 'T_Time', 'A_Dist', 'A_Time', 'BL_Dist',␣
 ↪'BL_Time',
                    'BM_Dist', 'BM_Time', 'BH_Dist', 'BH_Time', 'G_Dist',
                    'G_Time']

subset_df = df[selected_columns]
```

```
[11]: # Label Mapping
label_mapping = {
    2: 'Faster',
    1: 'Slower',
    0: 'Normal'
}

# Get value counts
counts = df['Target'].value_counts().sort_index()

# Use the labels from label_mapping for plotting
labels = [label_mapping[key] for key in counts.index]

# Plot the value counts
plt.bar(labels, counts)
plt.title('Altered Time Perception')
plt.xlabel('Time Perception')
plt.ylabel('Counts')
plt.show()
```



[12]:
```
# Examine Target value counts
subset_df['Target'].value_counts()
```

[12]:
```
2    61
1    33
0     7
Name: Target, dtype: int64
```

8 Subset Models

A Decision Tree classifier is a versatile machine learning algorithm that offers high interpretability by visualizing decision-making pathways. It can handle both numerical and categorical data, capture non-linear relationships, and requires no feature scaling.

[13]:
```
# Check for NANs
subset_df.isna().sum()
```

[13]:
```
Target     0
D_Dist     0
```

```
D_Time      0
T_Dist      0
T_Time      0
A_Dist      0
A_Time      0
BL_Dist     0
BL_Time     0
BM_Dist     0
BM_Time     0
BH_Dist     0
BH_Time     0
G_Dist      0
G_Time      0
dtype: int64
```

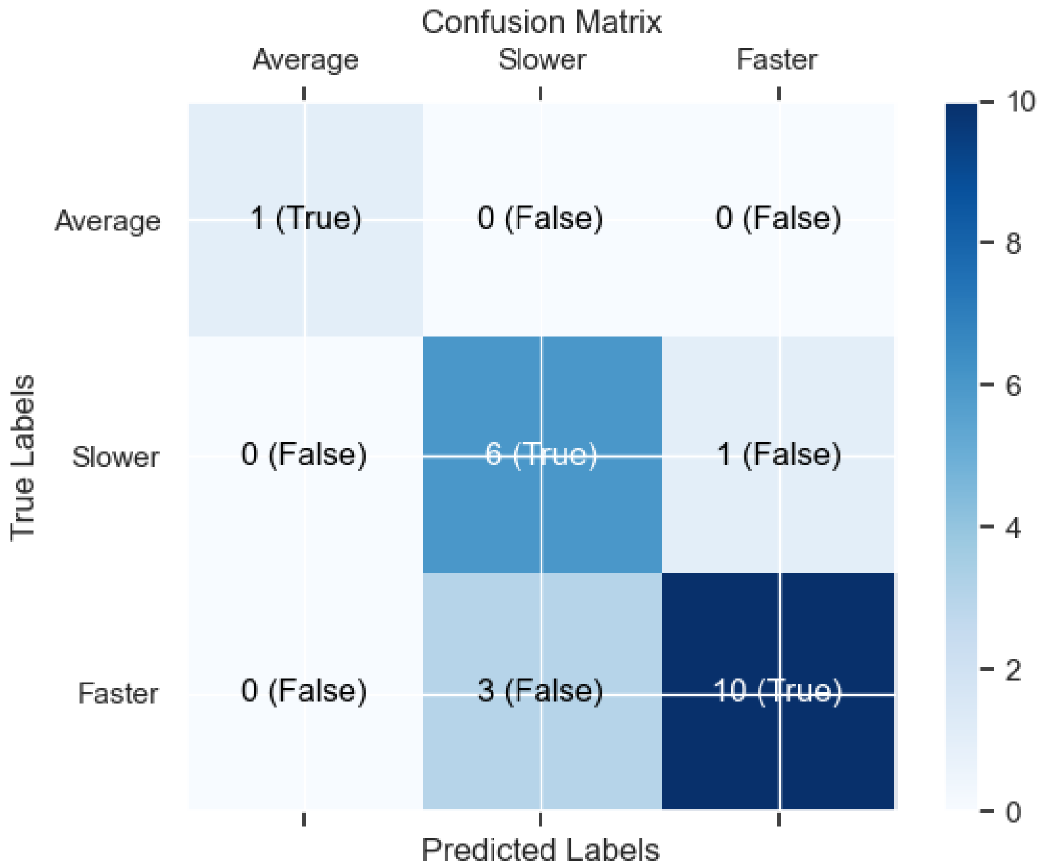
[14]:

```
# Split the data into training and test sets
X = subset_df.drop('Target', axis=1)
y = subset_df['Target']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
↪random_state=42, stratify=y)

# Initialize a Decision Tree classifier.
# Decision Trees are a non-parametric supervised learning method used for both
# classification and regression. It works by partitioning the source set into
# subsets based on the values of input attributes.
clf = DecisionTreeClassifier()

# Train the Decision Tree classifier on the training data.
# The fit method will construct a tree from the training data, trying to split
# on features and make decisions in a way that accurately predicts the target
# variable.
clf.fit(X_train, y_train)

# Once the model is trained, use it to predict the target variable for the test␣
↪data.
# The predict method traverses the trained decision tree to produce a␣
↪prediction
# for each test sample.
y_pred = clf.predict(X_test)

# Defining mapping of numeric labels to their corresponding word labels
numeric_labels = [0, 1, 2]
word_labels = ["Average", "Slower", "Faster"]

# Check accuracy
accuracy = accuracy_score(y_test, y_pred)*100
print()
```

```
print(f"Accuracy: {accuracy:.2f}%")
print()
print(classification_report(y_test, y_pred, target_names=word_labels))

# Feature names
feature_names = ['D_Dist', 'D_Time', 'T_Dist', 'T_Time', 'A_Dist', 'A_Time',␣
 ↪'BL_Dist',
                 'BL_Time', 'BM_Dist', 'BM_Time', 'BH_Dist', 'BH_Time',␣
 ↪'G_Dist', 'G_Time']

# Extracting feature importances from the Decision Tree model
feature_importances = clf.feature_importances_

# Pairing feature names with their importances and sorting them
sorted_importances = sorted(zip(feature_names, feature_importances), key=lambda␣
 ↪x: x[1], reverse=True)

# Display
print("\nFeature Importances:")
print()
for feature, importance in sorted_importances:
    print(f"{feature}: {importance:.4f}")

# Define numeric labels and corresponding word labels
numeric_labels = [0, 1, 2]
word_labels = ["Average", "Slower", "Faster"]

# Create a confusion matrix
cm = confusion_matrix(y_test, y_pred, labels=numeric_labels)

def plot_confusion_matrix(cm, labels):
    fig, ax = plt.subplots()
    cax = ax.matshow(cm, cmap=plt.cm.Blues)

    for i in range(cm.shape[0]):
        for j in range(cm.shape[1]):
            if i == j:
                color = "white" if cm[i, j] > cm.max() / 2 else "black"
                ax.text(j, i, f'{cm[i, j]} (True)', ha='center', va='center',␣
 ↪color=color)
            else:
                color = "white" if cm[i, j] > cm.max() / 2 else "black"
                ax.text(j, i, f'{cm[i, j]} (False)', ha='center', va='center',␣
 ↪color=color)

    fig.colorbar(cax)
    ax.set_xticks(np.arange(len(labels)))
```

```
    ax.set_yticks(np.arange(len(labels)))
    ax.set_xticklabels(labels)
    ax.set_yticklabels(labels)
    plt.xlabel('Predicted Labels')
    plt.ylabel('True Labels')
    plt.title('Confusion Matrix')
    plt.show()

# Use the existing plot function
plot_confusion_matrix(cm, word_labels)
```

```
Accuracy: 80.95%

              precision    recall  f1-score   support

     Average       1.00      1.00      1.00         1
      Slower       0.67      0.86      0.75         7
      Faster       0.91      0.77      0.83        13

    accuracy                           0.81        21
   macro avg       0.86      0.88      0.86        21
weighted avg       0.83      0.81      0.81        21


Feature Importances:

G_Dist: 0.2840
A_Dist: 0.2500
BL_Dist: 0.2295
BH_Dist: 0.1480
BM_Dist: 0.0648
BM_Time: 0.0236
D_Dist: 0.0000
D_Time: 0.0000
T_Dist: 0.0000
T_Time: 0.0000
A_Time: 0.0000
BL_Time: 0.0000
BH_Time: 0.0000
G_Time: 0.0000
```



[15]:
```
# Feature names for your dataset
feature_names = ['D_Dist', 'D_Time', 'T_Dist', 'T_Time', 'A_Dist', 'A_Time',␣
↪'BL_Dist',
                 'BL_Time', 'BM_Dist', 'BM_Time', 'BH_Dist', 'BH_Time',␣
↪'G_Dist', 'G_Time']

# Export the Decision Tree to a dot format
dot_data = export_graphviz(clf, out_file=None,
                           feature_names=feature_names,
                           class_names=word_labels,
                           filled=True, rounded=True,
                           special_characters=True)

# Use graphviz to create the graph object
graph = graphviz.Source(dot_data)

# View the graph
graph.view(filename="C:/Users/newmy/Desktop/TM_Project_Data_Files/
↪Decision_Tree", cleanup=True)
```

```
# Render and save the graph to a file (this is technically redundant after the␣
↪view command, but ensures the file is saved)
graph.render(filename="C:/Users/newmy/Desktop/TM_Project_Data_Files/
↪Decision_Tree", format='pdf', cleanup=True)
```

[15]: 'C:\\Users\\newmy\\Desktop\\TM_Project_Data_Files\\Decision_Tree.pdf'

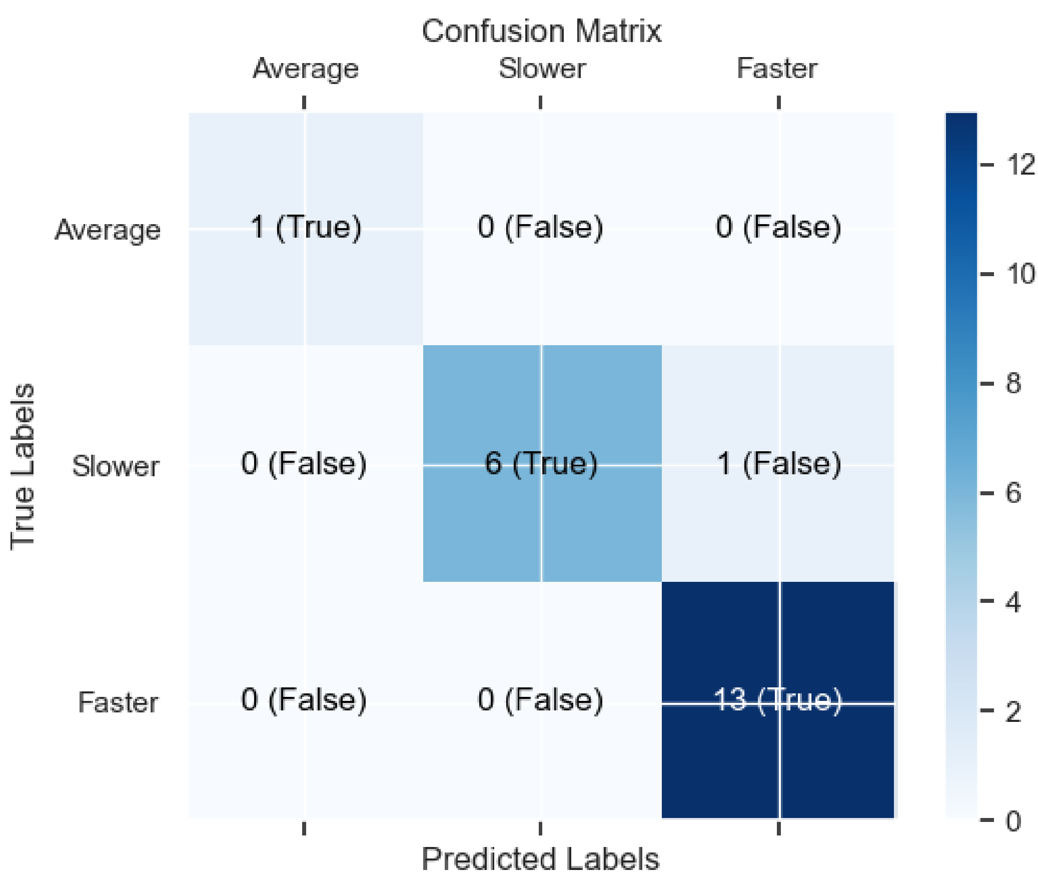
The following code is setting up and training a Random Forest classifier. A Random Forest is an ensemble learning method that creates a 'forest' of decision trees during training and outputs the class that is the mode of the classes (classification) or mean/average prediction (regression) of the individual trees for a given input.

[16]:
```
# Extracting features and target variable from the dataset
X = subset_df.drop('Target', axis=1)  # Features (excluding the target variable)
y = subset_df['Target']  # Target variable

# Splitting the data into training and testing sets, with 30% of the data being␣
↪used for testing
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.20,␣
↪random_state=42, stratify=y)

# Initializing a Random Forest classifier with 100 trees
clf = RandomForestClassifier(n_estimators=100, random_state=42)

# Training the Random Forest classifier on the training data
clf.fit(X_train, y_train)

# Predicting the target variable for the testing set
y_pred = clf.predict(X_test)

# Defining mapping of numeric labels to their corresponding word labels
numeric_labels = [0, 1, 2]
word_labels = ["Average", "Slower", "Faster", ]

accuracy_percentage = accuracy_score(y_test, y_pred) * 100
print(f"Accuracy: {accuracy_percentage:.2f}%")
print()
print(classification_report(y_test, y_pred, target_names=word_labels))

# Extract the feature importances
feature_importances = clf.feature_importances_

# Combine feature names and their importance scores
features_df = pd.DataFrame({
    'Feature': X_train.columns,
    'Importance': feature_importances
```

```
})

# Sort by importance
features_df = features_df.sort_values(by='Importance', ascending=False)

print(features_df.to_string(index=False))

# Create a confusion matrix
cm_ensemble = confusion_matrix(y_test, y_pred, labels=numeric_labels)

def plot_confusion_matrix(cm, labels):
    fig, ax = plt.subplots()
    cax = ax.matshow(cm, cmap=plt.cm.Blues)

    for i in range(cm.shape[0]):
        for j in range(cm.shape[1]):
            if i == j:
                color = "white" if cm[i, j] > cm.max() / 2 else "black"
                ax.text(j, i, f'{cm[i, j]} (True)', ha='center', va='center',␣
 ↪color=color)
            else:
                color = "white" if cm[i, j] > cm.max() / 2 else "black"
                ax.text(j, i, f'{cm[i, j]} (False)', ha='center', va='center',␣
 ↪color=color)

    fig.colorbar(cax)
    ax.set_xticks(np.arange(len(labels)))
    ax.set_yticks(np.arange(len(labels)))
    ax.set_xticklabels(labels)
    ax.set_yticklabels(labels)
    plt.xlabel('Predicted Labels')
    plt.ylabel('True Labels')
    plt.title('Confusion Matrix')
    plt.show()

# Use the existing plot function
plot_confusion_matrix(cm_ensemble, word_labels)
```

```
Accuracy: 95.24%

              precision    recall  f1-score   support

     Average       1.00      1.00      1.00         1
      Slower       1.00      0.86      0.92         7
      Faster       0.93      1.00      0.96        13

    accuracy                           0.95        21
```

```
   macro avg       0.98      0.95      0.96        21
weighted avg       0.96      0.95      0.95        21

Feature  Importance
 G_Dist    0.171840
 A_Dist    0.138343
BH_Dist    0.105299
BM_Dist    0.091901
BL_Dist    0.086646
BH_Time    0.061007
 D_Time    0.055371
 T_Time    0.050777
 D_Dist    0.050088
BL_Time    0.038846
 G_Time    0.038772
 T_Dist    0.038302
BM_Time    0.037629
 A_Time    0.035179
```



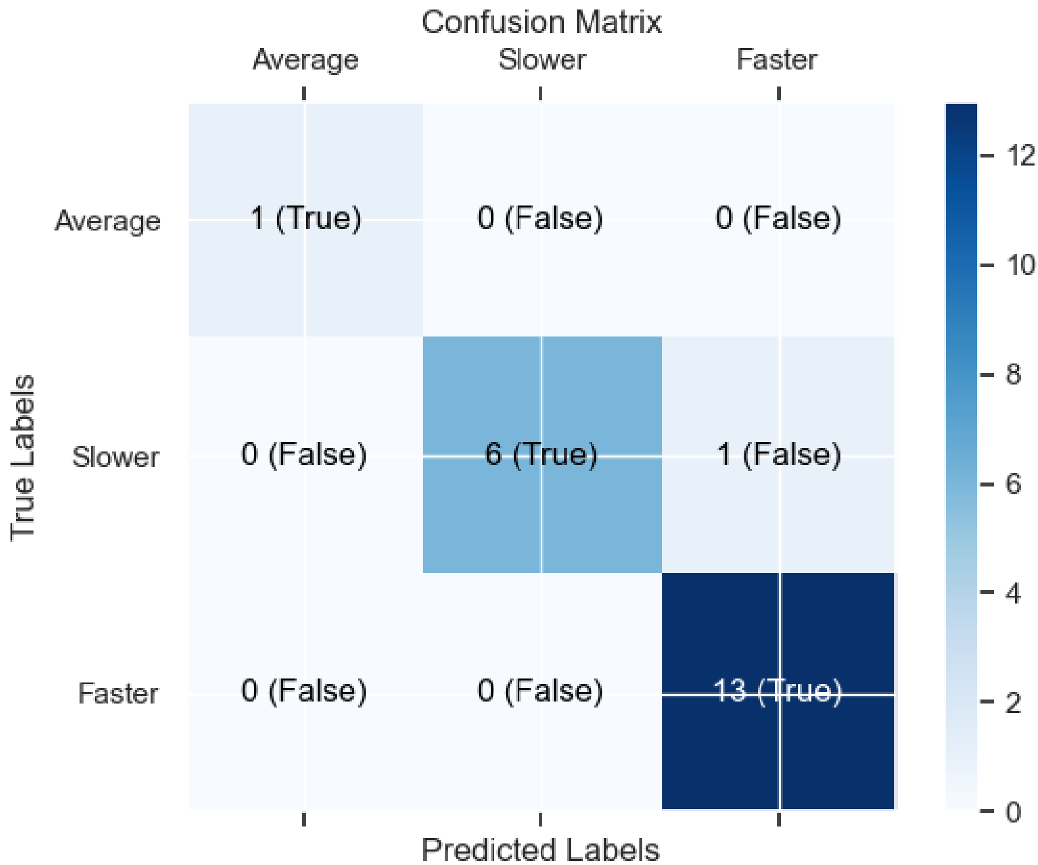
The following code is setting up and training a Random Forest classifier. However, unlike the previous code, it also includes a preprocessing step for standardizing features and uses a grid search to optimize hyperparameters for the Random Forest model.

[17]:
```
# Extracting features and target variable from the dataset
X = subset_df.drop('Target', axis=1)  # Features (excluding the target variable)
y = subset_df['Target']  # Target variable

# Splitting the data into training and testing sets. We use stratify to ensure
# the training and test datasets have similar proportion of target classes.
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, 
 ↪random_state=42, stratify=y)

# Standardizing the features. This step is optional for Random Forests since 
 ↪they are
# scale invariant, but is included for demonstration purposes.
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Initializing a base Random Forest classifier model
rf = RandomForestClassifier(random_state=42)

# Defining a grid of hyperparameters to optimize the Random Forest model
param_grid = {
    'n_estimators': [50, 100, 200],
    'max_depth': [None, 10, 20, 30],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4],
    'bootstrap': [True, False]
}

# Using GridSearchCV to search for the best hyperparameters over the specified 
 ↪grid.
# The search will be based on 3-fold cross-validation and will use all CPU 
 ↪cores (n_jobs=-1).
grid_search = GridSearchCV(estimator=rf, param_grid=param_grid,
                           cv=3, n_jobs=-1, verbose=2, scoring='accuracy')

# Training the model using GridSearchCV to find the best hyperparameters
grid_search.fit(X_train, y_train)

# Extracting the best Random Forest model after grid search
best_rf = grid_search.best_estimator_

# Predicting target variable for the test set using the best model
y_pred = best_rf.predict(X_test)
```

```
# Evaluating the model
accuracy_percentage = accuracy_score(y_test, y_pred) * 100
print()
print(f"Accuracy: {accuracy_percentage:.2f}%")
print()
print(classification_report(y_test, y_pred, target_names=word_labels))

# Extract the feature importances
feature_importances = best_rf.feature_importances_

# Combine feature names and their importance scores
features_df = pd.DataFrame({
    'Feature': X.columns,
    'Importance': feature_importances
})

# Sort by importance
features_df = features_df.sort_values(by='Importance', ascending=False)

print(features_df.to_string(index=False))

# Create a confusion matrix
cm_ensemble = confusion_matrix(y_test, y_pred, labels=numeric_labels)

def plot_confusion_matrix(cm, labels):
    fig, ax = plt.subplots()
    cax = ax.matshow(cm, cmap=plt.cm.Blues)

    for i in range(cm.shape[0]):
        for j in range(cm.shape[1]):
            if i == j:
                color = "white" if cm[i, j] > cm.max() / 2 else "black"
                ax.text(j, i, f'{cm[i, j]} (True)', ha='center', va='center',␣
↪color=color)
            else:
                color = "white" if cm[i, j] > cm.max() / 2 else "black"
                ax.text(j, i, f'{cm[i, j]} (False)', ha='center', va='center',␣
↪color=color)

    fig.colorbar(cax)
    ax.set_xticks(np.arange(len(labels)))
    ax.set_yticks(np.arange(len(labels)))
    ax.set_xticklabels(labels)
    ax.set_yticklabels(labels)
    plt.xlabel('Predicted Labels')
    plt.ylabel('True Labels')
```

```python
    plt.title('Confusion Matrix')
    plt.show()

# Use the existing plot function
plot_confusion_matrix(cm_ensemble, word_labels)
```

```
Fitting 3 folds for each of 216 candidates, totalling 648 fits

Accuracy: 95.24%

              precision    recall  f1-score   support

     Average       1.00      1.00      1.00         1
      Slower       1.00      0.86      0.92         7
      Faster       0.93      1.00      0.96        13

    accuracy                           0.95        21
   macro avg       0.98      0.95      0.96        21
weighted avg       0.96      0.95      0.95        21

Feature  Importance
 G_Dist    0.171840
 A_Dist    0.138343
BH_Dist    0.105299
BM_Dist    0.091901
BL_Dist    0.086646
BH_Time    0.061007
 D_Time    0.055371
 T_Time    0.050777
 D_Dist    0.050088
BL_Time    0.038846
 G_Time    0.038772
 T_Dist    0.038302
BM_Time    0.037629
 A_Time    0.035179
```



This code is preparing a dataset, splitting it into training and test subsets, pre-processing the data, initializing a Gradient Boosting Classifier, training it on the training data, and finally using the trained model to predict the target variable for the test data.

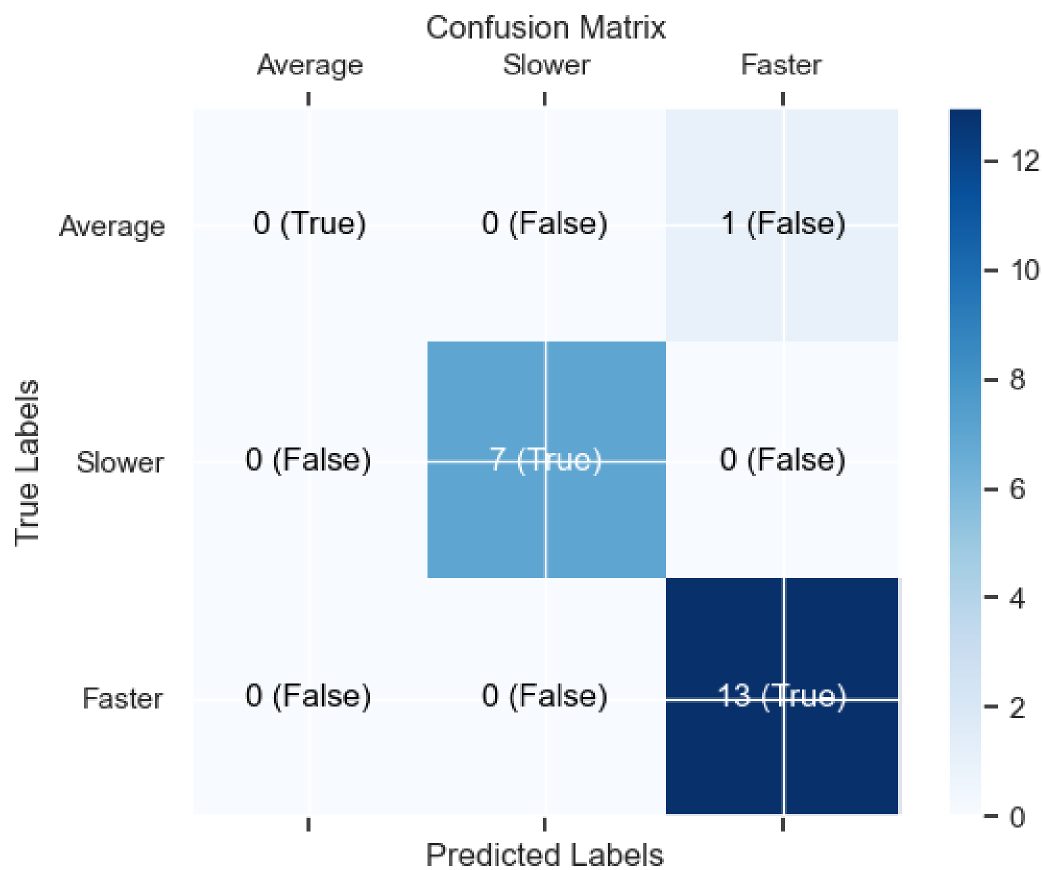
[18]:
```
# Drop the 'Target' column to get the feature matrix 'X'
X = subset_df.drop('Target', axis=1)

# Isolate the 'Target' column to get the target vector 'y'
y = subset_df['Target']

# Splitting the dataset into training and test sets.
# Using stratify ensures that the distribution of the target variable is␣
 ↪consistent between training and test sets.
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
 ↪random_state=42, stratify=y)

# Initialize a standard scaler. This will mean-center and scale the data to␣
 ↪unit variance.
```

```
# This step isn't necessary for Gradient Boosting since it's based on decision␣
↪trees, but
# sometimes it's used to maintain a consistent pre-processing pipeline or to␣
↪help with convergence.
scaler = StandardScaler()

# Fit the scaler on the training data and transform it.
X_train = scaler.fit_transform(X_train)

# Transform the test data using the same scaler (no fitting here to prevent␣
↪data leakage).
X_test = scaler.transform(X_test)

# Initialize a Gradient Boosting Classifier.
# n_estimators represents the number of boosting stages to be run.
# learning_rate shrinks the contribution of each tree.
# max_depth is the maximum depth of the individual trees.
gb = GradientBoostingClassifier(n_estimators=100, learning_rate=0.1,␣
↪max_depth=3, random_state=42)

# Train the Gradient Boosting Classifier on the training data.
gb.fit(X_train, y_train)

# Use the trained Gradient Boosting model to make predictions on the test data.
y_pred = gb.predict(X_test)

# Evaluating the model
accuracy_percentage = accuracy_score(y_test, y_pred) * 100
print(f"Accuracy: {accuracy_percentage:.2f}%")
print()

print(classification_report(y_test, y_pred, target_names=word_labels))

# Extract the feature importances
feature_importances = clf.feature_importances_

# Combine feature names and their importance scores
features_df = pd.DataFrame({
    'Feature': X.columns,
    'Importance': feature_importances
})

# Sort by importance
features_df = features_df.sort_values(by='Importance', ascending=False)

print(features_df.to_string(index=False))
```

```python
# Create a confusion matrix
cm_ensemble = confusion_matrix(y_test, y_pred, labels=numeric_labels)

def plot_confusion_matrix(cm, labels):
    fig, ax = plt.subplots()
    cax = ax.matshow(cm, cmap=plt.cm.Blues)

    for i in range(cm.shape[0]):
        for j in range(cm.shape[1]):
            if i == j:
                color = "white" if cm[i, j] > cm.max() / 2 else "black"
                ax.text(j, i, f'{cm[i, j]} (True)', ha='center', va='center',␣
 ↪color=color)
            else:
                color = "white" if cm[i, j] > cm.max() / 2 else "black"
                ax.text(j, i, f'{cm[i, j]} (False)', ha='center', va='center',␣
 ↪color=color)

    fig.colorbar(cax)
    ax.set_xticks(np.arange(len(labels)))
    ax.set_yticks(np.arange(len(labels)))
    ax.set_xticklabels(labels)
    ax.set_yticklabels(labels)
    plt.xlabel('Predicted Labels')
    plt.ylabel('True Labels')
    plt.title('Confusion Matrix')
    plt.show()

# Use the existing plot function
plot_confusion_matrix(cm_ensemble, word_labels)
```

```
Accuracy: 95.24%

              precision    recall  f1-score   support

     Average       0.00      0.00      0.00         1
      Slower       1.00      1.00      1.00         7
      Faster       0.93      1.00      0.96        13

    accuracy                           0.95        21
   macro avg       0.64      0.67      0.65        21
weighted avg       0.91      0.95      0.93        21

Feature  Importance
 G_Dist    0.171840
 A_Dist    0.138343
BH_Dist    0.105299
```

```
BM_Dist     0.091901
BL_Dist     0.086646
BH_Time     0.061007
 D_Time     0.055371
 T_Time     0.050777
 D_Dist     0.050088
BL_Time     0.038846
 G_Time     0.038772
 T_Dist     0.038302
BM_Time     0.037629
 A_Time     0.035179
```



This code demonstrates how to handle imbalanced datasets using SMOTE to generate synthetic instances of the minority class, and then trains a Random Forest Classifier on the balanced training set to make predictions on the test data.

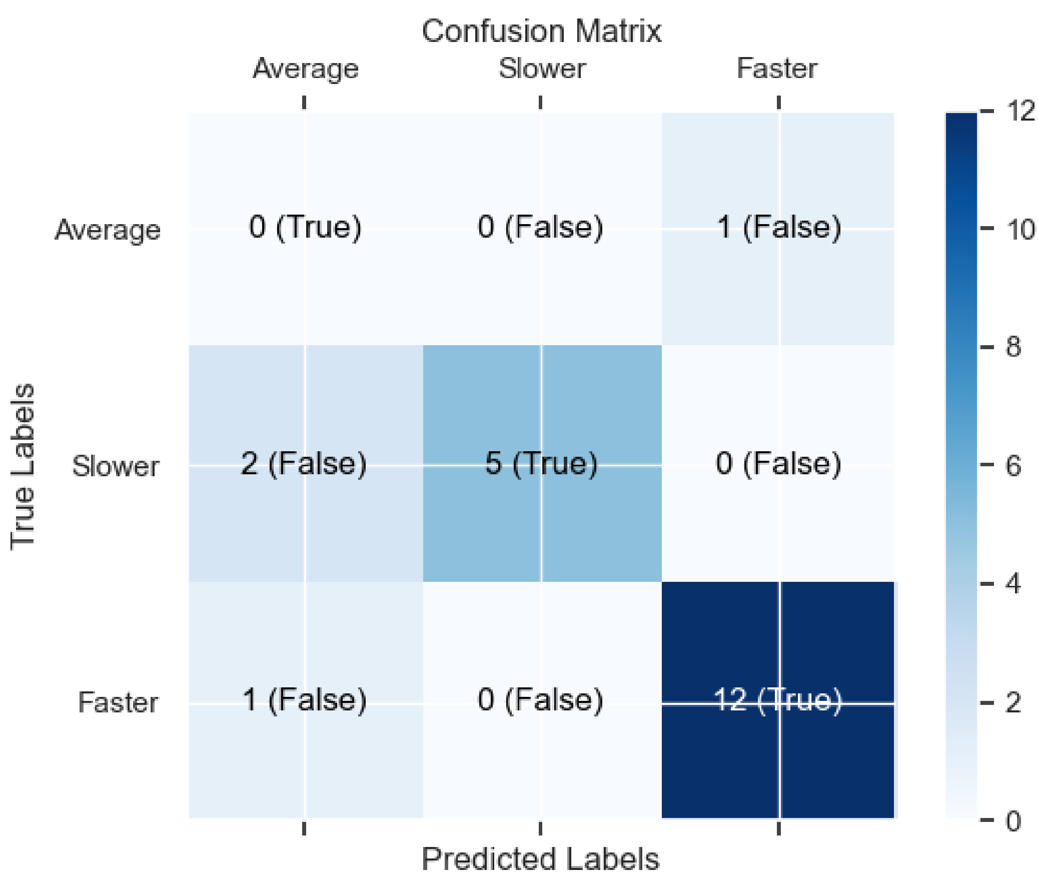
[19]:
```
# Extract features by dropping the 'Target' column, resulting in a features␣
↪matrix 'X'
X = subset_df.drop("Target", axis=1)
```

```
# Extract the 'Target' column to form the target vector 'y'
y = subset_df["Target"]

# Split the dataset into a training subset and a test subset
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.20,␣
 ↪random_state=42, stratify=y)

# SMOTE (Synthetic Minority Over-sampling Technique) is an over-sampling method␣
 ↪that creates synthetic examples
# in the feature space. It's used to handle imbalanced datasets by increasing␣
 ↪the number of instances in the minority class.
sm = SMOTE(k_neighbors=3)  # Using 3 nearest neighbors

# Apply SMOTE to the training data. This results in a balanced (or more␣
 ↪balanced) training dataset.
X_train_resampled, y_train_resampled = sm.fit_resample(X_train, y_train)

# Initialize a Random Forest Classifier. Random Forest is an ensemble learning␣
 ↪method
# that constructs multiple decision trees during training and outputs the␣
 ↪majority class
# (for classification problems) of the individual trees for predictions.
clf = RandomForestClassifier(n_estimators=100)  # Using 100 trees in the forest

# Train the Random Forest Classifier on the resampled (balanced) training data
clf.fit(X_train_resampled, y_train_resampled)

# Use the trained Random Forest model to predict the target for the test data
y_pred = clf.predict(X_test)

# Check accuracy
accuracy_percentage = accuracy_score(y_test, y_pred) * 100
print(f"Accuracy: {accuracy_percentage:.2f}%")
print()
print(classification_report(y_test, y_pred, target_names=word_labels))
print()
# Extract the feature importances
feature_importances = clf.feature_importances_

# Combine feature names and their importance scores
features_df = pd.DataFrame({
    'Feature': X.columns,
    'Importance': feature_importances
})

# Sort by importance
```

```python
features_df = features_df.sort_values(by='Importance', ascending=False)

print(features_df.to_string(index=False))
print()

# Create a confusion matrix
cm_ensemble = confusion_matrix(y_test, y_pred, labels=numeric_labels)

def plot_confusion_matrix(cm, labels):
    fig, ax = plt.subplots()
    cax = ax.matshow(cm, cmap=plt.cm.Blues)

    for i in range(cm.shape[0]):
        for j in range(cm.shape[1]):
            if i == j:
                color = "white" if cm[i, j] > cm.max() / 2 else "black"
                ax.text(j, i, f'{cm[i, j]} (True)', ha='center', va='center',␣
 ↪color=color)
            else:
                color = "white" if cm[i, j] > cm.max() / 2 else "black"
                ax.text(j, i, f'{cm[i, j]} (False)', ha='center', va='center',␣
 ↪color=color)

    fig.colorbar(cax)
    ax.set_xticks(np.arange(len(labels)))
    ax.set_yticks(np.arange(len(labels)))
    ax.set_xticklabels(labels)
    ax.set_yticklabels(labels)
    plt.xlabel('Predicted Labels')
    plt.ylabel('True Labels')
    plt.title('Confusion Matrix')
    plt.show()

# Use the existing plot function
plot_confusion_matrix(cm_ensemble, word_labels)
```

```
Accuracy: 80.95%

              precision    recall  f1-score   support

     Average       0.00      0.00      0.00         1
      Slower       1.00      0.71      0.83         7
      Faster       0.92      0.92      0.92        13

    accuracy                           0.81        21
   macro avg       0.64      0.55      0.59        21
weighted avg       0.90      0.81      0.85        21
```

```
Feature   Importance
 A_Dist    0.161328
 G_Dist    0.119011
BL_Dist    0.103028
BH_Dist    0.072968
 T_Dist    0.071484
 A_Time    0.071431
 T_Time    0.066114
BL_Time    0.060781
BM_Dist    0.059821
BH_Time    0.051667
 D_Time    0.046412
 G_Time    0.042446
 D_Dist    0.037689
BM_Time    0.035819
```



In essence, this code demonstrates the process of training a Random Forest model, extracting feature importances to identify which features are the most informative, and then re-training the model using only those top features to make predictions on the test data.

[20]:
```
# Initialize a Random Forest Classifier. Random Forest is an ensemble learning␣
↪method
# that constructs multiple decision trees during training and outputs the␣
↪majority class
# (for classification problems) of the individual trees for predictions.
clf = RandomForestClassifier(n_estimators=100)  # Using 100 trees in the forest

# Train the Random Forest Classifier on the training data
clf.fit(X_train, y_train)

# Obtain feature importances from the trained Random Forest model. This gives␣
↪insight
# into which features the model found to be the most informative for making␣
↪predictions.
feature_importances = clf.feature_importances_

# Convert the feature importances to a DataFrame for easier visualization and␣
↪sorting
features_df = pd.DataFrame({
    'Feature': X.columns,          # Feature names
    'Importance': feature_importances  # Their corresponding importance scores
})

# Display sorted feature importances
print("Feature Importances:")
print()
sorted_features = features_df.sort_values(by="Importance", ascending=False)
print(sorted_features.to_string(index=False))  # Display without the default␣
↪index for cleaner output

# Based on the sorted importances, select the top features.
# Here, we're assuming we want the top 10 most important features.
top_features = sorted_features['Feature'].head(10).tolist()

# Subset the training and test data to include only these top features
X_train_selected = X_train[top_features]
X_test_selected = X_test[top_features]

# Train the Random Forest Classifier using only the top features. This might␣
↪lead to a more focused and
# possibly better performing model if some features were not informative or␣
↪were introducing noise.
clf.fit(X_train_selected, y_train)
```

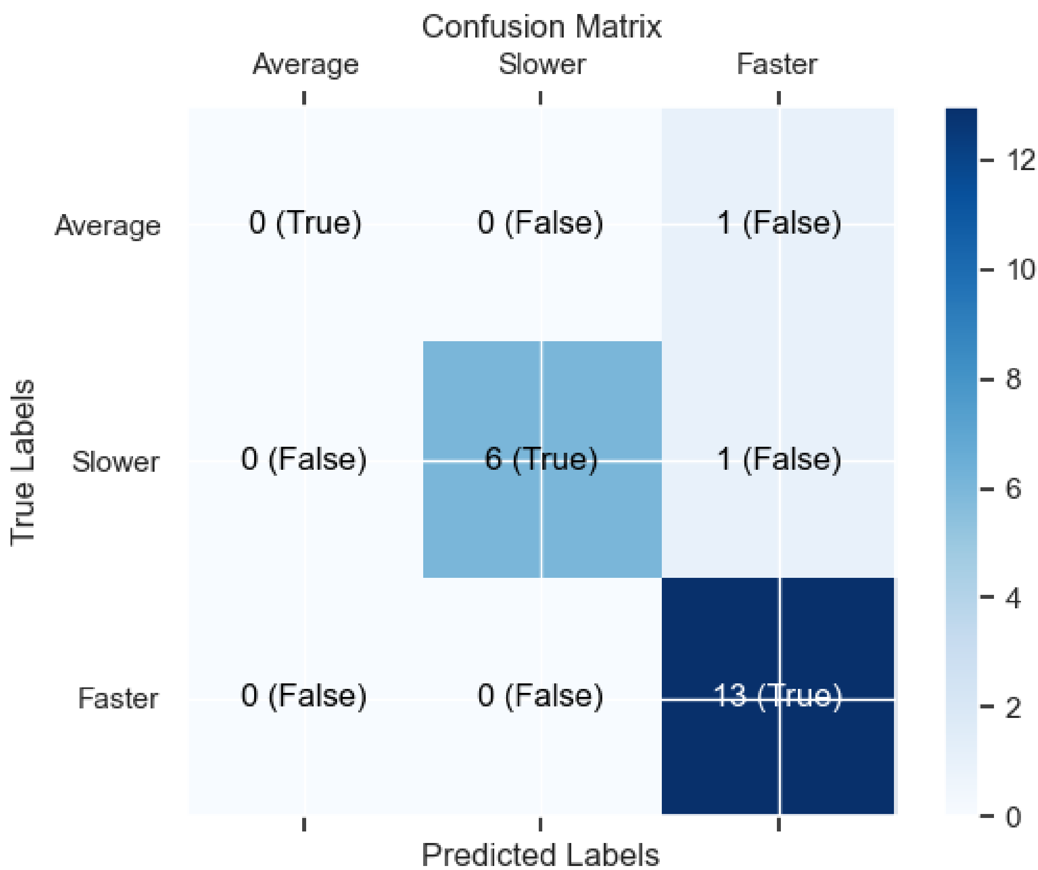
```python
# Use the re-trained Random Forest model to predict the target for the test data
y_pred = clf.predict(X_test_selected)

# Check accuracy
accuracy_percentage = accuracy_score(y_test, y_pred) * 100
print()
print(f"Accuracy: {accuracy_percentage:.2f}%")
print()
print(classification_report(y_test, y_pred, target_names=word_labels))
print()
# Define numeric labels and corresponding word labels
numeric_labels = [0, 1, 2]
word_labels = ["Average", "Slower", "Faster"]

# Create a confusion matrix
cm_ensemble = confusion_matrix(y_test, y_pred, labels=numeric_labels)

def plot_confusion_matrix(cm, labels):
    fig, ax = plt.subplots()
    cax = ax.matshow(cm, cmap=plt.cm.Blues)

    for i in range(cm.shape[0]):
        for j in range(cm.shape[1]):
            if i == j:
                color = "white" if cm[i, j] > cm.max() / 2 else "black"
                ax.text(j, i, f'{cm[i, j]} (True)', ha='center', va='center',␣
↪color=color)
            else:
                color = "white" if cm[i, j] > cm.max() / 2 else "black"
                ax.text(j, i, f'{cm[i, j]} (False)', ha='center', va='center',␣
↪color=color)

    fig.colorbar(cax)
    ax.set_xticks(np.arange(len(labels)))
    ax.set_yticks(np.arange(len(labels)))
    ax.set_xticklabels(labels)
    ax.set_yticklabels(labels)
    plt.xlabel('Predicted Labels')
    plt.ylabel('True Labels')
    plt.title('Confusion Matrix')
    plt.show()

# Use the existing plot function
plot_confusion_matrix(cm_ensemble, word_labels)
```

```
Feature Importances:
```

Feature	Importance
G_Dist	0.172940
A_Dist	0.149436
BL_Dist	0.103178
BM_Dist	0.091643
BH_Dist	0.090713
BH_Time	0.078437
D_Time	0.056550
D_Dist	0.048520
BL_Time	0.038840
BM_Time	0.037101
A_Time	0.036044
T_Dist	0.034695
G_Time	0.031196
T_Time	0.030709

Accuracy: 90.48%

	precision	recall	f1-score	support
Average	0.00	0.00	0.00	1
Slower	1.00	0.86	0.92	7
Faster	0.87	1.00	0.93	13
accuracy			0.90	21
macro avg	0.62	0.62	0.62	21
weighted avg	0.87	0.90	0.88	21



This code constructs an ensemble classifier, called a "Voting Classifier", using three individual models: a Random Forest, an SVM (Support Vector Machine), and a Logistic Regression. The ensemble classifier takes the predictions of each individual model and aggregates them to produce a final prediction. After training, the code computes the ensemble classifier's accuracy on test data.

[21]:
```
# Initialize individual models.

# RandomForest is an ensemble learning method that constructs multiple decision␣
↪trees
# during training and outputs the majority class (for classification problems)␣
↪of
# the individual trees for predictions.
clf1 = RandomForestClassifier(n_estimators=100)

# SVM is a supervised machine learning algorithm which can be used for␣
↪classification
# or regression problems. It uses a technique called the kernel trick to␣
↪transform
```

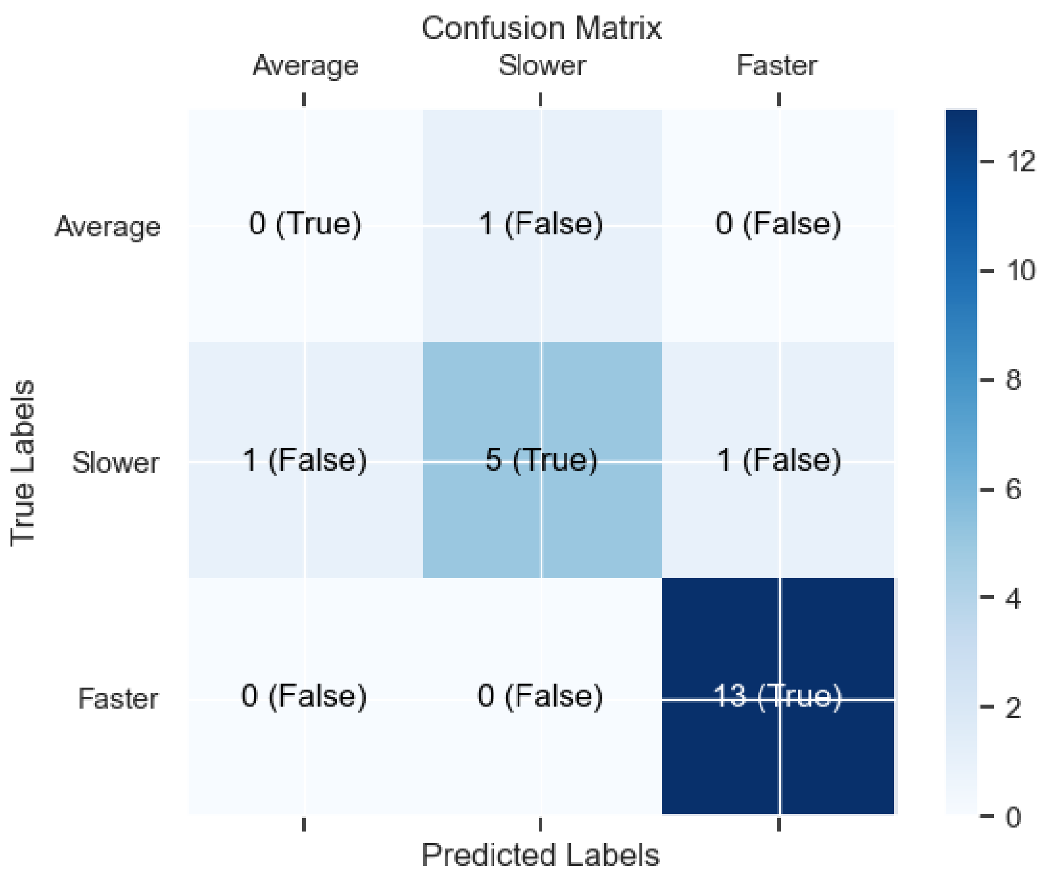
```
# your data and then based on these transformations it finds an optimal␣
↪boundary
# between the possible outputs.
clf2 = SVC(probability=True)  # probability=True allows to obtain probabilities␣
↪with predict_proba

# Logistic Regression is a statistical model that in its basic form uses a␣
↪logistic function
# to model a binary dependent variable.
clf3 = LogisticRegression()

# Create a voting classifier that combines the predictions from the three␣
↪individual models.
# The 'soft' voting means predictions are based on argmax of the sums of the␣
↪predicted
# probabilities, which recommends weighted average.
eclf = VotingClassifier(estimators=[
    ('rf', clf1), ('svc', clf2), ('lr', clf3)], voting='soft')

# Train the ensemble model on training data.
eclf.fit(X_train, y_train)

# Use the trained ensemble model to predict the target variable on the test␣
↪data.
y_pred = eclf.predict(X_test)

# Evaluate the ensemble model's performance.

# Calculate the accuracy of the ensemble model on the test data.
accuracy_ensemble = accuracy_score(y_test, y_pred)*100
print()
print(f"Ensemble Accuracy: {accuracy_ensemble:.2f}%")
print()

# Print a classification report showing various metrics (precision, recall,␣
↪f1-score, etc.)
classification_rep_ensemble = classification_report(y_test, y_pred,␣
↪target_names=word_labels)
print(classification_rep_ensemble)

# Extracting and combining feature importances from the models.

# List of feature names for reference.
feature_names = ['D_Dist', 'D_Time', 'T_Dist', 'T_Time', 'A_Dist', 'A_Time',␣
↪'BL_Dist',
```

```python
                 'BL_Time', 'BM_Dist', 'BM_Time', 'BH_Dist', 'BH_Time', 'G_Dist', 'G_Time']

# Access each individual model inside the VotingClassifier after training.
fitted_rf = eclf.named_estimators_['rf']
fitted_svc = eclf.named_estimators_['svc']
fitted_lr = eclf.named_estimators_['lr']

# Extract feature importances from RandomForest.
feature_importances_rf = fitted_rf.feature_importances_

# Extract feature importances from SVM.
# Note: Importances from SVM are relevant only when the SVM uses a linear kernel.
if isinstance(fitted_svc.kernel, str) and fitted_svc.kernel == 'linear':
    feature_importances_svc = abs(fitted_svc.coef_[0])
else:
    feature_importances_svc = np.ones(len(feature_names))  # Assign uniform importance for non-linear SVM.

# Extract feature importances from Logistic Regression.
feature_importances_lr = abs(fitted_lr.coef_[0])

# Helper function to normalize the feature importances.
def normalize(importance):
    return importance / sum(importance)

# Normalize the feature importances for each model.
normalized_rf = normalize(feature_importances_rf)
normalized_svc = normalize(feature_importances_svc)
normalized_lr = normalize(feature_importances_lr)

# Combine (sum) normalized importances from all models.
combined_importance = normalized_rf + normalized_svc + normalized_lr

# Pair the feature names with their combined importances and sort them in descending order.
sorted_importances = sorted(zip(feature_names, combined_importance), key=lambda x: x[1], reverse=True)
# Display
print("Feature Importances:")
print()
for feature, importance in sorted_importances:
    print(f"{feature}: {importance:.4f}")

# Define numeric labels and corresponding word labels
numeric_labels = [0, 1, 2]
```

```
word_labels = ["Average", "Slower", "Faster"]

# Create a confusion matrix
cm_ensemble = confusion_matrix(y_test, y_pred, labels=numeric_labels)

def plot_confusion_matrix(cm, labels):
    fig, ax = plt.subplots()
    cax = ax.matshow(cm, cmap=plt.cm.Blues)

    for i in range(cm.shape[0]):
        for j in range(cm.shape[1]):
            if i == j:
                color = "white" if cm[i, j] > cm.max() / 2 else "black"
                ax.text(j, i, f'{cm[i, j]} (True)', ha='center', va='center',␣
 ↪color=color)
            else:
                color = "white" if cm[i, j] > cm.max() / 2 else "black"
                ax.text(j, i, f'{cm[i, j]} (False)', ha='center', va='center',␣
 ↪color=color)

    fig.colorbar(cax)
    ax.set_xticks(np.arange(len(labels)))
    ax.set_yticks(np.arange(len(labels)))
    ax.set_xticklabels(labels)
    ax.set_yticklabels(labels)
    plt.xlabel('Predicted Labels')
    plt.ylabel('True Labels')
    plt.title('Confusion Matrix')
    plt.show()

# Use the existing plot function
plot_confusion_matrix(cm_ensemble, word_labels)
```

```
Ensemble Accuracy: 85.71%

              precision    recall  f1-score   support

     Average       0.00      0.00      0.00         1
      Slower       0.83      0.71      0.77         7
      Faster       0.93      1.00      0.96        13

    accuracy                           0.86        21
   macro avg       0.59      0.57      0.58        21
weighted avg       0.85      0.86      0.85        21

Feature Importances:
```

```
BL_Time: 0.3996
BM_Time: 0.3784
BH_Time: 0.3067
A_Dist: 0.2472
A_Time: 0.2451
G_Dist: 0.2108
BM_Dist: 0.1872
BH_Dist: 0.1854
BL_Dist: 0.1722
G_Time: 0.1605
T_Dist: 0.1375
D_Time: 0.1358
D_Dist: 0.1229
T_Time: 0.1108
```



9 Full Dataset Models

[22]:
```
selected_columns = ['Target', 'Myelin', 'D_WL',
                    'D_Cycle', 'D_Dist', 'D_Time', 'D_Freq', 'T_WL', 'T_Cycle',
                    'T_Dist', 'T_Time', 'T_Freq', 'A_WL', 'A_Cycle', 'A_Dist',
                    'A_Time', 'A_Freq', 'BL_WL', 'BL_Cycle', 'BL_Dist', 'BL_Time',
                    'BL_Freq', 'BM_WL', 'BM_Cycle', 'BM_Dist', 'BM_Time',␣
 ↪'BM_Freq',
                    'BH_WL', 'BH_Cycle', 'BH_Dist', 'BH_Time', 'BH_Freq', 'G_WL',
                    'G_Freq', 'G_Dist', 'G_Time']

large_subset_df = df[selected_columns]
```

[23]:
```
feature_names = large_subset_df.columns.tolist()
print(feature_names)
```

```
['Target', 'Myelin', 'D_WL', 'D_Cycle', 'D_Dist', 'D_Time', 'D_Freq', 'T_WL',
'T_Cycle', 'T_Dist', 'T_Time', 'T_Freq', 'A_WL', 'A_Cycle', 'A_Dist', 'A_Time',
'A_Freq', 'BL_WL', 'BL_Cycle', 'BL_Dist', 'BL_Time', 'BL_Freq', 'BM_WL',
'BM_Cycle', 'BM_Dist', 'BM_Time', 'BM_Freq', 'BH_WL', 'BH_Cycle', 'BH_Dist',
'BH_Time', 'BH_Freq', 'G_WL', 'G_Freq', 'G_Dist', 'G_Time']
```

[24]:
```
if 'Target' in df.columns:
    print("Column 'Target' exists!")
else:
    print("Column 'Target' does not exist!")
```

```
Column 'Target' exists!
```

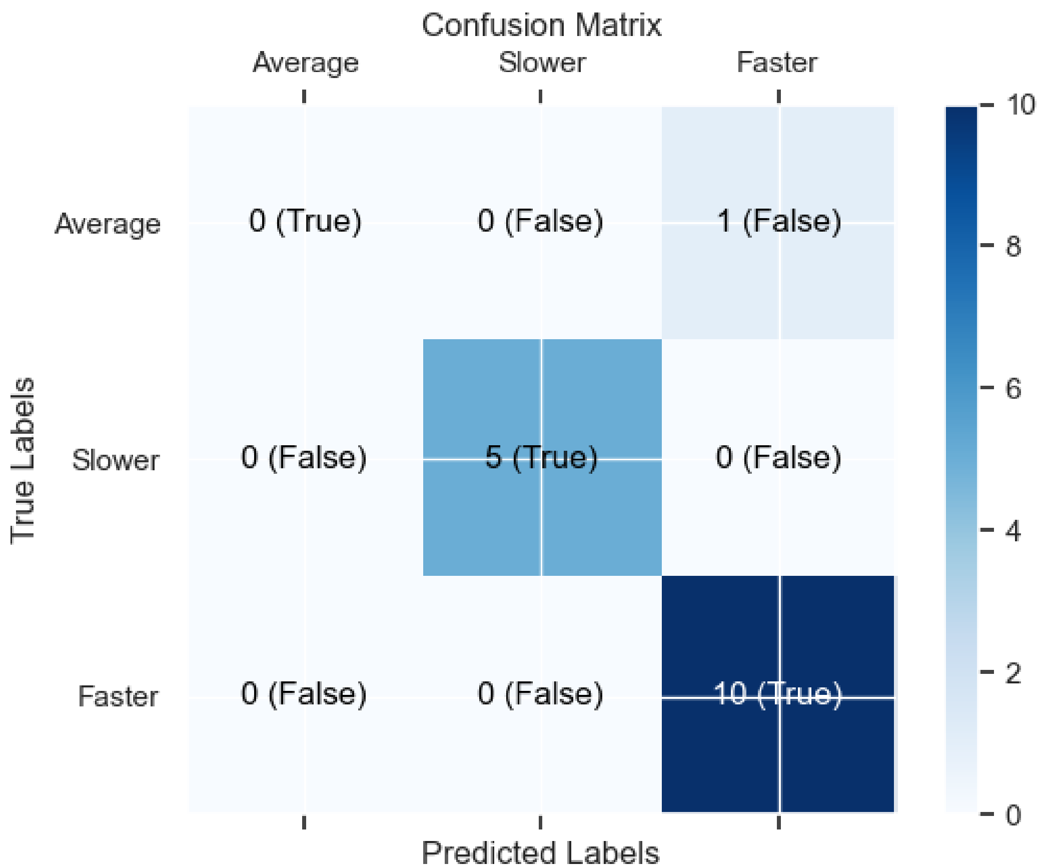
[25]:
```
# Filter the dataframe to only include columns with non-object datatypes
large_subset_df = large_subset_df.select_dtypes(exclude=['object'])

# Extract features by dropping the 'Target' column
X = large_subset_df.drop('Target', axis=1)

# Extract the target variable 'Target'
y = large_subset_df['Target']

# Split the dataset into training (70%) and testing (30%) sets using a␣
 ↪consistent random seed for reproducibility
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.15,␣
 ↪random_state=42, stratify=y)

# Initialize a RandomForestClassifier with 100 trees and a consistent random␣
 ↪seed for reproducibility
clf = RandomForestClassifier(n_estimators=100, random_state=42)

# Train the RandomForest classifier on the training data
```

```python
clf.fit(X_train, y_train)

# Use the trained classifier to predict the target variable for the test set
y_pred = clf.predict(X_test)

# Define numeric labels and corresponding word labels
numeric_labels = [0, 1, 2]
word_labels = ["Average", "Slower", "Faster"]

accuracy_percentage = accuracy_score(y_test, y_pred) * 100
print(f"Accuracy: {accuracy_percentage:.2f}%")
print()
print(classification_report(y_test, y_pred, target_names=word_labels))

# Extract the feature importances
feature_importances = clf.feature_importances_

# Combine feature names and their importance scores
features_df = pd.DataFrame({
    'Feature': X_train.columns,
    'Importance': feature_importances
})

# Sort by importance
features_df = features_df.sort_values(by='Importance', ascending=False)

print(features_df.to_string(index=False))

# Create a confusion matrix
cm_ensemble = confusion_matrix(y_test, y_pred, labels=numeric_labels)

def plot_confusion_matrix(cm, labels):
    fig, ax = plt.subplots()
    cax = ax.matshow(cm, cmap=plt.cm.Blues)

    for i in range(cm.shape[0]):
        for j in range(cm.shape[1]):
            if i == j:
                color = "white" if cm[i, j] > cm.max() / 2 else "black"
                ax.text(j, i, f'{cm[i, j]} (True)', ha='center', va='center',␣
 ↪color=color)
            else:
                color = "white" if cm[i, j] > cm.max() / 2 else "black"
                ax.text(j, i, f'{cm[i, j]} (False)', ha='center', va='center',␣
 ↪color=color)

    fig.colorbar(cax)
```

```
    ax.set_xticks(np.arange(len(labels)))
    ax.set_yticks(np.arange(len(labels)))
    ax.set_xticklabels(labels)
    ax.set_yticklabels(labels)
    plt.xlabel('Predicted Labels')
    plt.ylabel('True Labels')
    plt.title('Confusion Matrix')
    plt.show()

# Use the existing plot function
plot_confusion_matrix(cm_ensemble, word_labels)
```

```
Accuracy: 93.75%

              precision    recall  f1-score   support

     Average       0.00      0.00      0.00         1
      Slower       1.00      1.00      1.00         5
      Faster       0.91      1.00      0.95        10

    accuracy                           0.94        16
   macro avg       0.64      0.67      0.65        16
weighted avg       0.88      0.94      0.91        16

 Feature  Importance
  Myelin    0.216234
   BM_WL    0.171053
   BL_WL    0.113611
   BH_WL    0.074452
 D_Cycle    0.043163
 BH_Freq    0.034247
  G_Dist    0.033419
    A_WL    0.031802
    D_WL    0.028705
  A_Dist    0.022372
 BL_Dist    0.020708
  D_Freq    0.017337
    G_WL    0.015877
 BH_Dist    0.015533
  G_Freq    0.014742
 BM_Dist    0.011600
  G_Time    0.011359
 A_Cycle    0.010109
 BL_Freq    0.008932
 BL_Time    0.008901
 BM_Freq    0.008852
BM_Cycle    0.008798
BH_Cycle    0.008389
```

```
  T_Cycle     0.008383
   T_Freq     0.008373
   T_Time     0.008088
   A_Freq     0.007766
 BL_Cycle     0.007390
  BM_Time     0.006461
   D_Time     0.006400
  BH_Time     0.005107
     T_WL     0.003444
   A_Time     0.003386
   T_Dist     0.003126
   D_Dist     0.001881
```



[26]:
```python
print(len(feature_importances_rf))
print(len(feature_importances_svc))
print(len(feature_importances_lr))
```

```
14
14
14
```

[27]:
```
feature_names = X.columns.tolist()
print(feature_names)
```

```
['Myelin', 'D_WL', 'D_Cycle', 'D_Dist', 'D_Time', 'D_Freq', 'T_WL', 'T_Cycle',
'T_Dist', 'T_Time', 'T_Freq', 'A_WL', 'A_Cycle', 'A_Dist', 'A_Time', 'A_Freq',
'BL_WL', 'BL_Cycle', 'BL_Dist', 'BL_Time', 'BL_Freq', 'BM_WL', 'BM_Cycle',
'BM_Dist', 'BM_Time', 'BM_Freq', 'BH_WL', 'BH_Cycle', 'BH_Dist', 'BH_Time',
'BH_Freq', 'G_WL', 'G_Freq', 'G_Dist', 'G_Time']
```

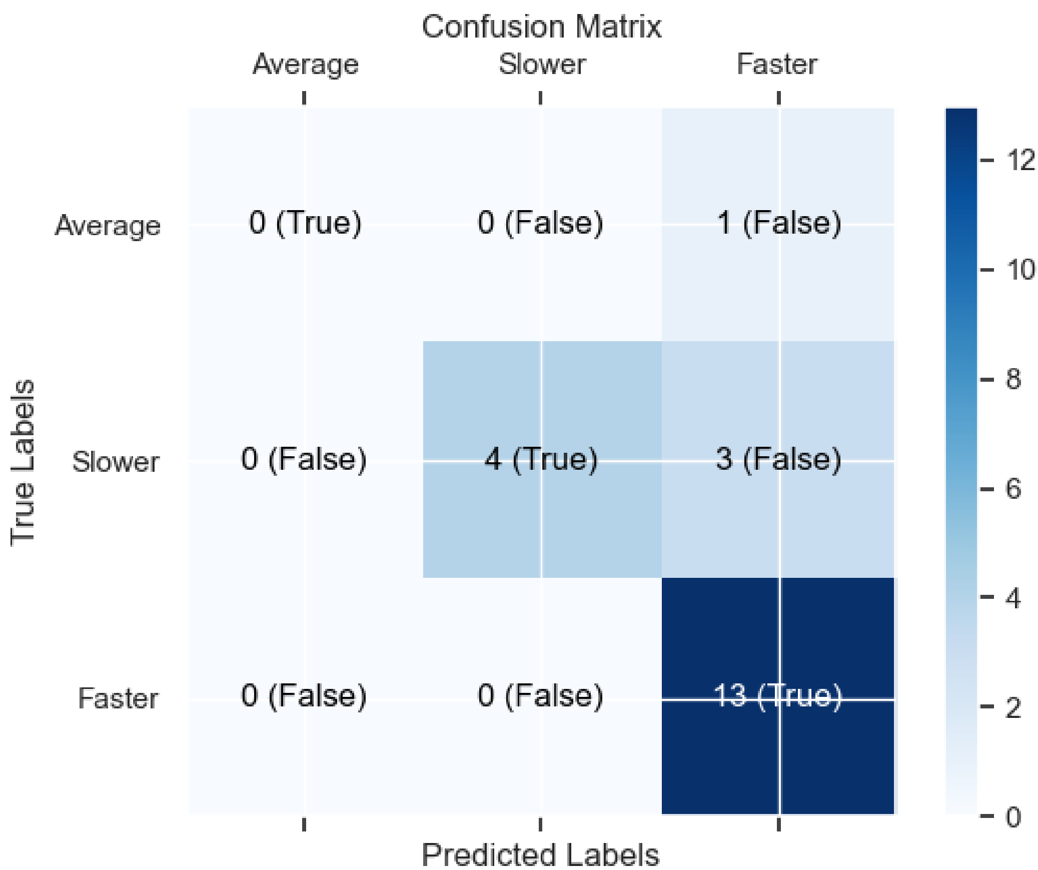
[28]:
```
# Split the data into features and target
X = large_subset_df.drop('Target', axis=1)
y = large_subset_df['Target']

# Split the data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.20,␣
 ↪random_state=42, stratify=y)

# Define individual models
clf1 = RandomForestClassifier(n_estimators=100)
clf2 = SVC(probability=True)
clf3 = LogisticRegression()

# Create a voting classifier
eclf = VotingClassifier(estimators=[
    ('rf', clf1), ('svc', clf2), ('lr', clf3)], voting='soft')

# Train the ensemble model
eclf.fit(X_train, y_train)

# Predict on test data
y_pred = eclf.predict(X_test)

# 1. Check accuracy
# Calculate and print accuracy and classification report for the ensemble
accuracy_ensemble = accuracy_score(y_test, y_pred)*100
print()
print(f"Ensemble Accuracy: {accuracy_ensemble:.2f}%")
print()
classification_rep_ensemble = classification_report(y_test, y_pred,␣
 ↪target_names=word_labels)
print(classification_rep_ensemble)


# Extracting feature importances

# Feature names
```

```
feature_names = ['Myelin', 'D_WL', 'D_Cycle', 'D_Dist', 'D_Time', 'D_Freq',␣
↪'T_WL',
                 'T_Cycle', 'T_Dist', 'T_Time', 'T_Freq', 'A_WL', 'A_Cycle',␣
↪'A_Dist',
                 'A_Time', 'A_Freq', 'BL_WL', 'BL_Cycle', 'BL_Dist', 'BL_Time',
                 'BL_Freq', 'BM_WL', 'BM_Cycle', 'BM_Dist', 'BM_Time',␣
↪'BM_Freq',
                 'BH_WL', 'BH_Cycle', 'BH_Dist', 'BH_Time', 'BH_Freq', 'G_WL',
                 'G_Freq', 'G_Dist', 'G_Time']

# Access the fitted models within the VotingClassifier
fitted_rf = eclf.named_estimators_['rf']
fitted_svc = eclf.named_estimators_['svc']
fitted_lr = eclf.named_estimators_['lr']

# RandomForest
feature_importances_rf = fitted_rf.feature_importances_

# SVM (assuming a linear kernel)
if isinstance(fitted_svc.kernel, str) and fitted_svc.kernel == 'linear':
    feature_importances_svc = abs(fitted_svc.coef_[0])
else:
    feature_importances_svc = np.ones(len(feature_names))  # Placeholder for␣
↪non-linear SVM

# Logistic Regression
feature_importances_lr = abs(fitted_lr.coef_[0])

# Normalize function
def normalize(importance):
    return importance / sum(importance)

# Normalizing the importances
normalized_rf = normalize(feature_importances_rf)
normalized_svc = normalize(feature_importances_svc)
normalized_lr = normalize(feature_importances_lr)

# Combining the normalized scores
combined_importance = normalized_rf + normalized_svc + normalized_lr

# Pairing feature names with their importances and sorting them
sorted_importances = sorted(zip(feature_names, combined_importance), key=lambda␣
↪x: x[1], reverse=True)

# Display
print("Feature Importances:")
print()
```

```python
for feature, importance in sorted_importances:
    print(f"{feature}: {importance:.4f}")

# Define numeric labels and corresponding word labels
numeric_labels = [0, 1, 2]
word_labels = ["Average", "Slower", "Faster"]

# Create a confusion matrix
cm_ensemble = confusion_matrix(y_test, y_pred, labels=numeric_labels)

def plot_confusion_matrix(cm, labels):
    fig, ax = plt.subplots()
    cax = ax.matshow(cm, cmap=plt.cm.Blues)

    for i in range(cm.shape[0]):
        for j in range(cm.shape[1]):
            if i == j:
                color = "white" if cm[i, j] > cm.max() / 2 else "black"
                ax.text(j, i, f'{cm[i, j]} (True)', ha='center', va='center',␣
↪color=color)
            else:
                color = "white" if cm[i, j] > cm.max() / 2 else "black"
                ax.text(j, i, f'{cm[i, j]} (False)', ha='center', va='center',␣
↪color=color)

    fig.colorbar(cax)
    ax.set_xticks(np.arange(len(labels)))
    ax.set_yticks(np.arange(len(labels)))
    ax.set_xticklabels(labels)
    ax.set_yticklabels(labels)
    plt.xlabel('Predicted Labels')
    plt.ylabel('True Labels')
    plt.title('Confusion Matrix')
    plt.show()

# Use the existing plot function
plot_confusion_matrix(cm_ensemble, word_labels)
```

```
Ensemble Accuracy: 80.95%

              precision    recall  f1-score   support

     Average       0.00      0.00      0.00         1
      Slower       1.00      0.57      0.73         7
      Faster       0.76      1.00      0.87        13
```

```
    accuracy                           0.81        21
   macro avg       0.59      0.52      0.53        21
weighted avg       0.81      0.81      0.78        21

Feature Importances:

Myelin: 0.2826
D_Dist: 0.2329
BL_Dist: 0.2028
T_Cycle: 0.1836
BM_WL: 0.1819
T_Dist: 0.1698
BM_Dist: 0.1580
D_Cycle: 0.1340
BL_WL: 0.1334
BH_WL: 0.1117
BH_Dist: 0.0945
A_Cycle: 0.0843
A_WL: 0.0716
A_Dist: 0.0714
G_Dist: 0.0667
G_Freq: 0.0591
D_WL: 0.0555
BH_Cycle: 0.0532
D_Freq: 0.0453
BM_Cycle: 0.0449
BH_Freq: 0.0446
G_WL: 0.0443
BL_Cycle: 0.0433
BH_Time: 0.0382
T_Time: 0.0382
BL_Freq: 0.0371
A_Freq: 0.0368
T_WL: 0.0365
BM_Freq: 0.0362
G_Time: 0.0354
A_Time: 0.0352
BL_Time: 0.0352
BM_Time: 0.0344
D_Time: 0.0343
T_Freq: 0.0332
```



[29]:
```
# Assuming 'y' contains classes like 'Average', 'Slower', 'Faster'

# Identify unique classes and sort them if necessary
unique_classes = y.unique()
unique_classes.sort()  # Only if you want them sorted

# Create word_labels
word_labels = unique_classes.tolist()  # ['Average', 'Slower', 'Faster']

# Create numeric_labels
numeric_labels = list(range(len(unique_classes)))  # [0, 1, 2]

# Print to verify
print("Word Labels:", word_labels)
print("Numeric Labels:", numeric_labels)
```

```
Word Labels: [0, 1, 2]
Numeric Labels: [0, 1, 2]
```

[30]:

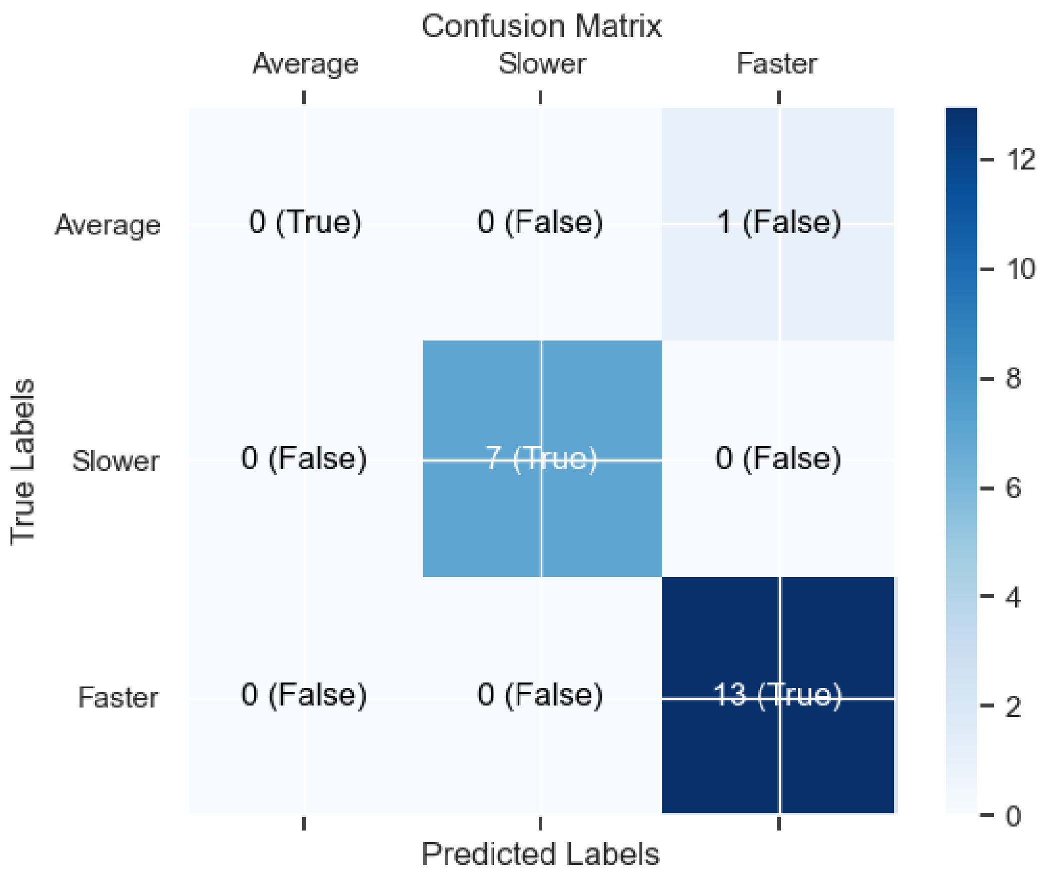
```python
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, classification_report,␣
↪confusion_matrix
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np

# Split data into features and target variable
X = large_subset_df.drop('Target', axis=1)
y = large_subset_df['Target']

# Train-test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.20,␣
↪random_state=42, stratify=y)

# Initialize RandomForestClassifier
clf = RandomForestClassifier(random_state=42)

# Define the parameter grid
param_grid = {
    'n_estimators': [300, 500, 800],
    'max_depth': [None, 20, 40, 60],
    'min_samples_split': [2, 4, 6],
    'min_samples_leaf': [1, 2, 3],
    'max_features': ['sqrt', 'log2']
}

# Initialize GridSearchCV
grid_search = GridSearchCV(clf, param_grid, cv=10, verbose=2, n_jobs=-1)

# Fit GridSearchCV to the training data
grid_search.fit(X_train, y_train)

# Get the best estimator
best_clf = grid_search.best_estimator_

# Predict on the test data
y_pred = best_clf.predict(X_test)

print("Best Parameters:", grid_search.best_params_)

# Define numeric labels and corresponding word labels
numeric_labels = [0, 1, 2]
word_labels = ["Average", "Slower", "Faster"]

accuracy_percentage = accuracy_score(y_test, y_pred) * 100
```

```
print(f"Accuracy: {accuracy_percentage:.2f}%")
print()
print(classification_report(y_test, y_pred, target_names=word_labels))

# Extract the feature importances from the best estimator
feature_importances = best_clf.feature_importances_

# Combine feature names and their importance scores
features_df = pd.DataFrame({
    'Feature': X_train.columns,
    'Importance': feature_importances
})

# Sort by importance
features_df = features_df.sort_values(by='Importance', ascending=False)

print(features_df.to_string(index=False))

# Create a confusion matrix
cm_ensemble = confusion_matrix(y_test, y_pred, labels=numeric_labels)

def plot_confusion_matrix(cm, labels):
    fig, ax = plt.subplots()
    cax = ax.matshow(cm, cmap=plt.cm.Blues)

    for i in range(cm.shape[0]):
        for j in range(cm.shape[1]):
            if i == j:
                color = "white" if cm[i, j] > cm.max() / 2 else "black"
                ax.text(j, i, f'{cm[i, j]} (True)', ha='center', va='center',␣
↪color=color)
            else:
                color = "white" if cm[i, j] > cm.max() / 2 else "black"
                ax.text(j, i, f'{cm[i, j]} (False)', ha='center', va='center',␣
↪color=color)

    fig.colorbar(cax)
    ax.set_xticks(np.arange(len(labels)))
    ax.set_yticks(np.arange(len(labels)))
    ax.set_xticklabels(labels)
    ax.set_yticklabels(labels)
    plt.xlabel('Predicted Labels')
    plt.ylabel('True Labels')
    plt.title('Confusion Matrix')
    plt.show()

# Use the existing plot function
```

```
plot_confusion_matrix(cm_ensemble, word_labels)
```

```
Fitting 10 folds for each of 216 candidates, totalling 2160 fits
Best Parameters: {'max_depth': None, 'max_features': 'sqrt', 'min_samples_leaf':
1, 'min_samples_split': 2, 'n_estimators': 300}
Accuracy: 95.24%

              precision    recall  f1-score   support

     Average       0.00      0.00      0.00         1
      Slower       1.00      1.00      1.00         7
      Faster       0.93      1.00      0.96        13

    accuracy                           0.95        21
   macro avg       0.64      0.67      0.65        21
weighted avg       0.91      0.95      0.93        21

 Feature  Importance
  Myelin    0.219146
   BM_WL    0.138901
   BL_WL    0.130158
   BH_WL    0.082301
  G_Dist    0.042733
 D_Cycle    0.036264
    A_WL    0.036033
  A_Dist    0.025721
    G_WL    0.021063
  D_Freq    0.018391
 BM_Dist    0.018085
 BH_Freq    0.017949
 BL_Dist    0.017858
    D_WL    0.016627
  G_Freq    0.015839
 BH_Dist    0.014632
BM_Cycle    0.012719
BH_Cycle    0.012635
 A_Cycle    0.010921
 BM_Freq    0.009833
BL_Cycle    0.009166
    T_WL    0.008463
  T_Time    0.007952
 T_Cycle    0.007799
  T_Freq    0.007761
  A_Freq    0.007266
  D_Dist    0.007145
 BL_Freq    0.006419
 BM_Time    0.006333
  D_Time    0.006091
```

```
  G_Time     0.006071
 BL_Time     0.006005
  A_Time     0.005794
  T_Dist     0.005121
 BH_Time     0.004805
```



[31]:
```
print(large_subset_df.columns)
```

```
Index(['Target', 'Myelin', 'D_WL', 'D_Cycle', 'D_Dist', 'D_Time', 'D_Freq',
       'T_WL', 'T_Cycle', 'T_Dist', 'T_Time', 'T_Freq', 'A_WL', 'A_Cycle',
       'A_Dist', 'A_Time', 'A_Freq', 'BL_WL', 'BL_Cycle', 'BL_Dist', 'BL_Time',
       'BL_Freq', 'BM_WL', 'BM_Cycle', 'BM_Dist', 'BM_Time', 'BM_Freq',
       'BH_WL', 'BH_Cycle', 'BH_Dist', 'BH_Time', 'BH_Freq', 'G_WL', 'G_Freq',
       'G_Dist', 'G_Time'],
      dtype='object')
```

[32]:
```
from sklearn.model_selection import GridSearchCV
from sklearn.ensemble import RandomForestClassifier

# Split data into features and target variable
```

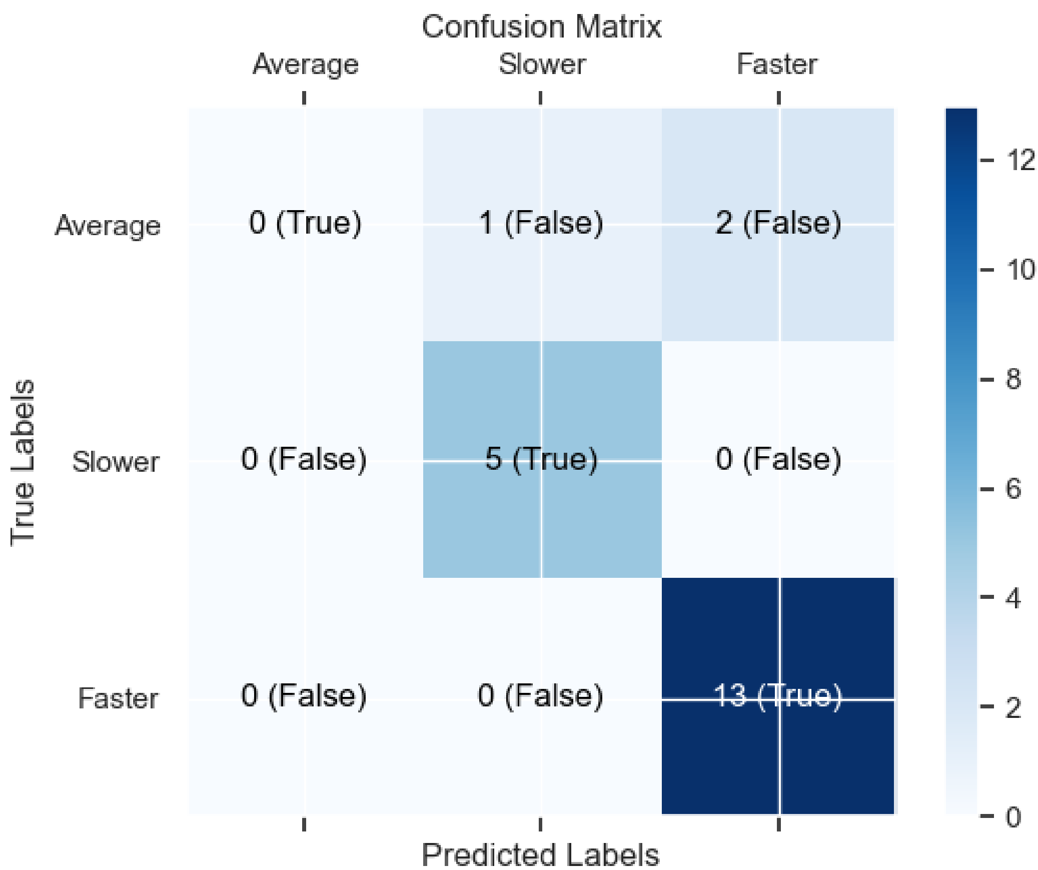
```
X = large_subset_df.drop('Target', axis=1)
y = large_subset_df['Target']

# Train-test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.20,␣
↪random_state=42)

# Initialize RandomForestClassifier
clf = RandomForestClassifier(random_state=42)

# Define an expanded parameter grid
param_grid = {
    'n_estimators': [250, 500, 750],       # More fine-grained choice
    'max_depth': [10, 20, 30, 40, 50],     # Added a smaller depth to avoid␣
↪potential overfitting
    'min_samples_split': [2, 4, 6],
    'min_samples_leaf': [1, 2, 3],
    'max_features': ['sqrt', 'log2'],
    'bootstrap': [True, False]             # Added bootstrap option
}

# Initialize GridSearchCV
grid_search = GridSearchCV(clf, param_grid, cv=10, verbose=2, n_jobs=-1)

# Fit GridSearchCV to the training data
grid_search.fit(X_train, y_train)

# Print the best parameters
print("Best Parameters:", grid_search.best_params_)

# Assuming you want to test the best model against your test data
best_clf = grid_search.best_estimator_
y_pred = best_clf.predict(X_test)

accuracy_percentage = accuracy_score(y_test, y_pred) * 100
print(f"Accuracy: {accuracy_percentage:.2f}%")
print()
print(classification_report(y_test, y_pred, target_names=word_labels))

# Extract the feature importances from the best estimator
feature_importances = best_clf.feature_importances_

# Combine feature names and their importance scores
features_df = pd.DataFrame({
    'Feature': X_train.columns,
    'Importance': feature_importances
})
```

```python
# Sort by importance
features_df = features_df.sort_values(by='Importance', ascending=False)

print(features_df.to_string(index=False))

# Create a confusion matrix
cm_ensemble = confusion_matrix(y_test, y_pred, labels=numeric_labels)

def plot_confusion_matrix(cm, labels):
    fig, ax = plt.subplots()
    cax = ax.matshow(cm, cmap=plt.cm.Blues)

    for i in range(cm.shape[0]):
        for j in range(cm.shape[1]):
            if i == j:
                color = "white" if cm[i, j] > cm.max() / 2 else "black"
                ax.text(j, i, f'{cm[i, j]} (True)', ha='center', va='center',␣
 ↪color=color)
            else:
                color = "white" if cm[i, j] > cm.max() / 2 else "black"
                ax.text(j, i, f'{cm[i, j]} (False)', ha='center', va='center',␣
 ↪color=color)

    fig.colorbar(cax)
    ax.set_xticks(np.arange(len(labels)))
    ax.set_yticks(np.arange(len(labels)))
    ax.set_xticklabels(labels)
    ax.set_yticklabels(labels)
    plt.xlabel('Predicted Labels')
    plt.ylabel('True Labels')
    plt.title('Confusion Matrix')
    plt.show()

# Use the existing plot function
plot_confusion_matrix(cm_ensemble, word_labels)
```

```
Fitting 10 folds for each of 540 candidates, totalling 5400 fits
Best Parameters: {'bootstrap': True, 'max_depth': 10, 'max_features': 'sqrt',
'min_samples_leaf': 1, 'min_samples_split': 2, 'n_estimators': 500}
Accuracy: 85.71%

              precision    recall  f1-score   support

     Average       0.00      0.00      0.00         3
      Slower       0.83      1.00      0.91         5
      Faster       0.87      1.00      0.93        13
```

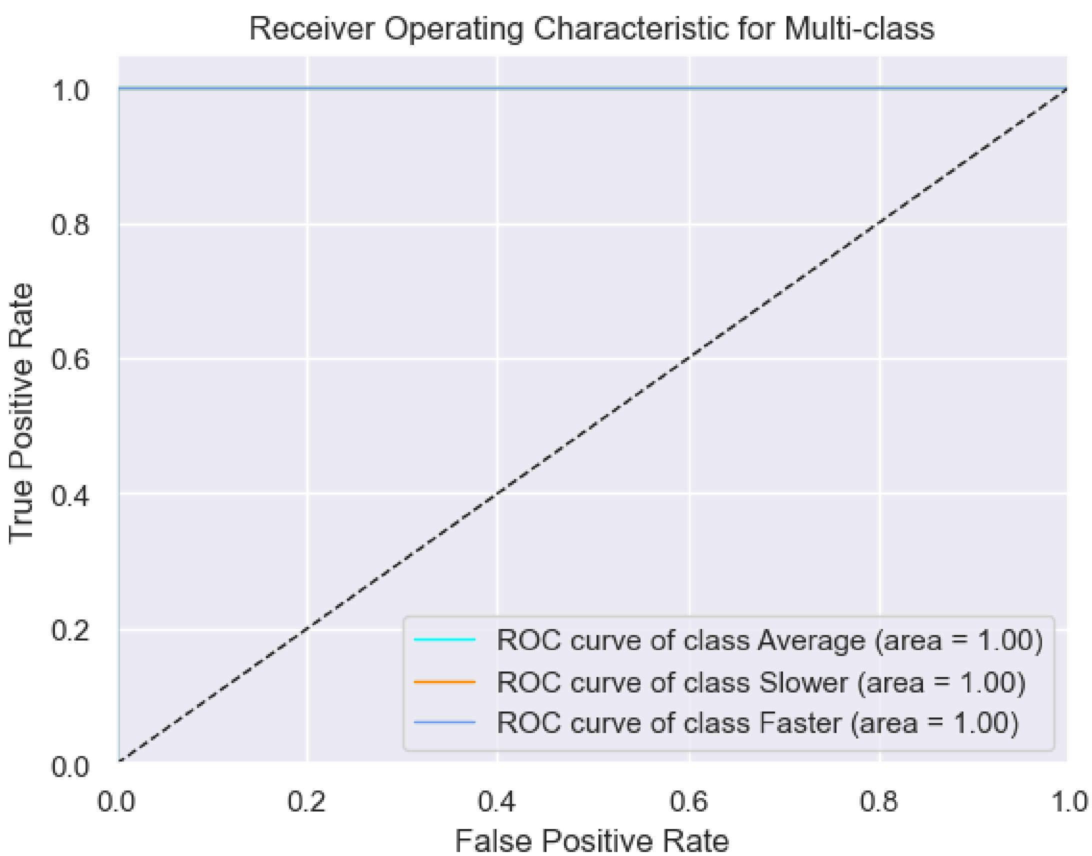
accuracy			0.86	21
macro avg	0.57	0.67	0.61	21
weighted avg	0.73	0.86	0.79	21

Feature	Importance
Myelin	0.209635
BM_WL	0.127776
BL_WL	0.112719
BH_WL	0.082339
G_Dist	0.055631
A_WL	0.039210
D_Cycle	0.035956
A_Dist	0.031880
G_Freq	0.031087
D_WL	0.027120
D_Freq	0.020508
G_WL	0.019046
A_Cycle	0.018262
BH_Freq	0.015113
BM_Dist	0.014930
A_Freq	0.014921
BL_Dist	0.014488
BH_Cycle	0.012775
BH_Dist	0.011941
BM_Cycle	0.010408
BL_Freq	0.009476
BM_Freq	0.009449
T_Freq	0.008905
G_Time	0.007726
T_WL	0.007463
T_Cycle	0.006788
A_Time	0.006634
BM_Time	0.006537
D_Time	0.005762
D_Dist	0.005214
BH_Time	0.004964
BL_Cycle	0.004901
T_Time	0.004504
T_Dist	0.003655
BL_Time	0.002278



[33]:
```
# Visualize the ROC curves

from sklearn.metrics import roc_curve, auc
from sklearn.preprocessing import label_binarize
from itertools import cycle

# Class labels
class_labels = ['Average','Slower','Faster']

# Binarize the output for multiclass ROC curve
y_bin = label_binarize(y_test, classes=[0, 1, 2])
y_prob = grid_search.predict_proba(X_test)
n_classes = y_bin.shape[1]

# Compute ROC curve and ROC area for each class
fpr = dict()
tpr = dict()
roc_auc = dict()
for i in range(n_classes):
```

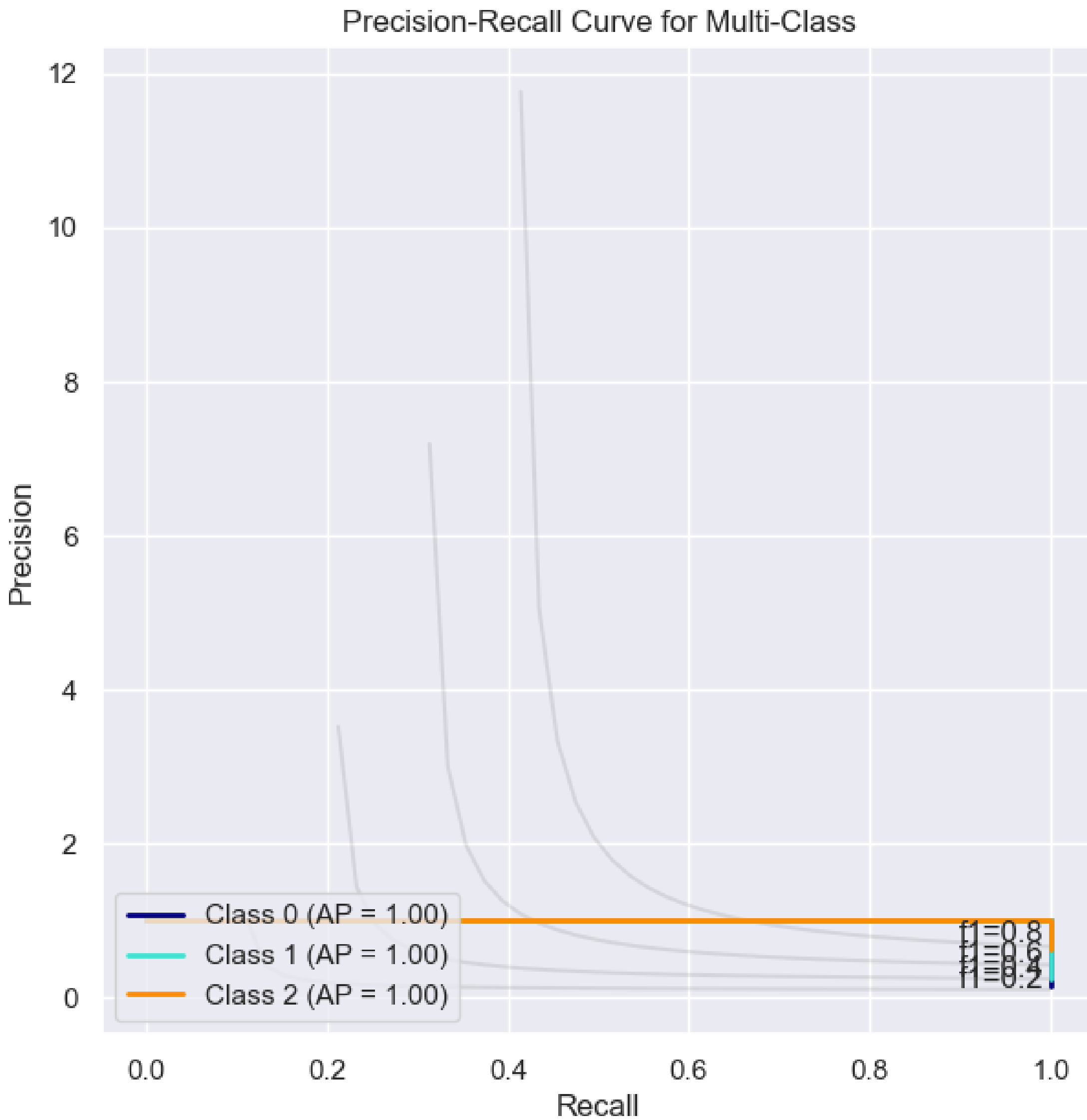
```
    fpr[i], tpr[i], _ = roc_curve(y_bin[:, i], y_prob[:, i])
    roc_auc[i] = auc(fpr[i], tpr[i])

# Plot all ROC curves
plt.figure()
colors = cycle(['aqua', 'darkorange', 'cornflowerblue', 'green'])
for i, color in zip(range(n_classes), colors):
    plt.plot(fpr[i], tpr[i], color=color, lw=1, label='ROC curve of class {0}␣
 ↪(area = {1:0.2f})'.format(class_labels[i], roc_auc[i]))

plt.plot([0, 1], [0, 1], 'k--', lw=1)
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic for Multi-class')
plt.legend(loc="lower right")
plt.show()
```



[35]:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.preprocessing import label_binarize
from sklearn.metrics import precision_recall_curve, average_precision_score
from itertools import cycle

# Assuming you have a multiclass classifier 'model' and a dataset (X_test,␣
↪y_test)
# y_test should be the true class labels
# Classes should be labeled as 0 to n_classes-1
n_classes = len(np.unique(y_test))

# Binarize the output (one-hot encoding)
y_test_binarized = label_binarize(y_test, classes=np.arange(n_classes))

# Getting prediction probabilities
y_score = grid_search.predict_proba(X_test)

# Compute Precision-Recall and average precision for each class
precision = dict()
recall = dict()
average_precision = dict()

for i in range(n_classes):
    precision[i], recall[i], _ = precision_recall_curve(y_test_binarized[:, i],␣
↪y_score[:, i])
    average_precision[i] = average_precision_score(y_test_binarized[:, i],␣
↪y_score[:, i])

# Plot Precision-Recall curve for each class and iso-f1 curves
plt.figure(figsize=(7, 7))
colors = cycle(['navy', 'turquoise', 'darkorange', 'cornflowerblue', 'teal'])

f_scores = np.linspace(0.2, 0.8, num=4)
for f_score in f_scores:
    x = np.linspace(0.01, 1)
    y = f_score * x / (2 * x - f_score)
    l, = plt.plot(x[y >= 0], y[y >= 0], color='gray', alpha=0.2)
    plt.annotate('f1={0:0.1f}'.format(f_score), xy=(0.9, y[45] + 0.02))

for i, color in zip(range(n_classes), colors):
    plt.plot(recall[i], precision[i], color=color, lw=2,
             label='Class {0} (AP = {1:0.2f})'.format(i, average_precision[i]))

plt.xlabel('Recall')
plt.ylabel('Precision')
plt.title('Precision-Recall Curve for Multi-Class')
```

```
plt.legend(loc="lower left")
plt.show()
```



10 Conclusion

In this machine learning project, we have employed Random Forest classifiers in conjunction with Grid Search for model tuning, leveraging two distinct subsets of data. The first subset, smaller in scale, focused on ‘distance’ and ‘time’ features. The second, more comprehensive dataset incorporated a broader range of features. Throughout the model tuning phase, both datasets yielded commendable performance, with accuracy rates oscillating between 80.95% and 95.24% across different model runs. This range indicates a robust adaptability of the models to varying data complexities and volumes.

A key aspect of our analysis involved addressing data imbalance, a common challenge in machine learning projects. The positive trends observed on both the ROC (Receiver Operating Characteristic) and precision-recall curves suggest that our models were effectively trained with this imbalance in mind. These curves are critical in evaluating the trade-off between true positive rates and false positives, and their encouraging results bolster confidence in the model's reliability.

Moreover, the development of 'Cammie_r', a mathematical predictor designed as a target variable, demonstrated valid accuracy predictions. Its effectiveness in this initial phase suggests its potential utility in larger datasets. The successful implementation and results of 'Cammie_r' lay a solid foundation for its application in the subsequent part of our project, 'Temporal Metrics'.

In conclusion, the achieved results are not only promising but also applicable for use in the continuation of this project. The methodologies and insights gained form a concrete basis for further exploration and development in the next phase. The combination of Random Forest and Grid Search techniques, coupled with careful consideration of data characteristics, has proven effective, and we can proceed with confidence in the expansion of our work to include more extensive datasets and refined predictive modeling.

```
[ ]:
```