Bright-Hope Mood Disorders Data Analysis | Group 2

01 - Import Required Libraries

In [2]:
```
# Data Manipulation
import pandas as pd
import numpy as np

# Data Visualization
import matplotlib.pyplot as plt
import seaborn as sns

# Machine Learning Processing
from sklearn.preprocessing import OrdinalEncoder, StandardScaler
from sklearn.preprocessing import OneHotEncoder, MinMaxScaler

# Machine Learning Modeling
from sklearn.compose import ColumnTransformer
from sklearn.ensemble import RandomForestClassifier, ExtraTreesClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.tree import DecisionTreeClassifier

# Machine Learning Evaluation
from sklearn.model_selection import train_test_split
from sklearn.model_selection import cross_val_score, GridSearchCV, KFold
from sklearn.metrics import confusion_matrix
from sklearn.metrics import accuracy_score, precision_score, recall_score
from imblearn.metrics import specificity_score
```

02 - Load Mood Disorders Dataset

In [4]:
```
# Explicitly setting skip_blank_lines=True did not prevent
# blank rows with NaN values; dropped them separately instead.
try:
    df_mood = pd.read_csv('MoodDisorders.csv', dtype=object).dropna(how='all')
except FileNotFoundError: print("ERROR - DATA FILE NOT FOUND")
except pd.errors.EmptyDataError: print("ERROR - NO DATA IN FILE")
except pd.errors.ParserError: print("ERROR - PARSING")
except Exception: print("ERROR - SOMETHING WENT WRONG")

df_mood.head()
```

Out[4]:

	PersonNum	Sadness	Euphoria	Exhaustion	Sleeplessness	MoodSwing	SuicidalThoughts	Anorxia
0	Person-1	Usually	Seldom	Sometimes	Sometimes	YES	YES	NO
1	Person-2	Usually	Seldom	Usually	Sometimes	NO	YES	NO
2	Person-3	Sometimes	Most-Often	Sometimes	Sometimes	YES	NO	NO
3	Person-4	Usually	Seldom	Usually	Most-Often	YES	YES	YES
4	Person-5	Usually	Usually	Sometimes	Sometimes	NO	NO	NO

03 - Data Pre-Processing

Dimensions & Features

In [5]:
```
df_mood.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 240 entries, 0 to 239
Data columns (total 19 columns):
 #   Column            Non-Null Count  Dtype 
---  ------            --------------  ----- 
 0   PersonNum         240 non-null    object
 1   Sadness           240 non-null    object
 2   Euphoria          240 non-null    object
 3   Exhaustion        240 non-null    object
 4   Sleeplessness     240 non-null    object
 5   MoodSwing         240 non-null    object
 6   SuicidalThoughts  240 non-null    object
 7   Anorxia           240 non-null    object
 8   Disobedience      240 non-null    object
 9   JustifyBehavior   240 non-null    object
 10  Aggressiveness    240 non-null    object
 11  MoveOn            240 non-null    object
 12  NervousBreakdown  240 non-null    object
 13  AdmitMistakes     240 non-null    object
 14  Overthinking      240 non-null    object
 15  SexualActivity    240 non-null    object
 16  Concentration     240 non-null    object
 17  Optimisim         240 non-null    object
 18  Diagnosis         240 non-null    object
dtypes: object(19)
memory usage: 37.5+ KB
```

Scanning for Missing Values

In [6]:
```
df_mood.isnull().sum()
```

Out[6]:
```
PersonNum           0
Sadness             0
Euphoria            0
Exhaustion          0
Sleeplessness       0
MoodSwing           0
SuicidalThoughts    0
Anorxia             0
Disobedience        0
JustifyBehavior     0
Aggressiveness      0
MoveOn              0
NervousBreakdown    0
AdmitMistakes       0
Overthinking        0
SexualActivity      0
Concentration       0
Optimisim           0
Diagnosis           0
dtype: int64
```

Scanning for Duplicate Rows

In [7]:
```
duplicate_rows = df_mood.duplicated()

if duplicate_rows.any():
    print(f"""
        Duplicate rows found:
        {df_mood[duplicate_rows]}
    """)
else:
    print("No duplicate rows found.")
```

```
No duplicate rows found.
```

Fixing `PersonNum` as the PK

In [8]:
```
# First, check that each `PersonNum` is unique
# to validate as the primary key.
duplicate_vals = df_mood['PersonNum'].duplicated()

if duplicate_vals.any():
    print(f"""
        Duplicate rows found:
        {df_mood[duplicate_vals]}
    """)
else:
    print("No duplicate values found.")
```

```
No duplicate values found.
```

In [9]:
```
# Adjust the values in the 'PersonNum' column
# to retain only the ID number
df_mood['PersonNum'] = df_mood['PersonNum'].str.split('-').str[-1]
df_mood['PersonNum'] = df_mood['PersonNum'].astype(int)

df_mood = df_mood.set_index('PersonNum')

df_mood.head()
```

Out[9]:

PersonNum	Sadness	Euphoria	Exhaustion	Sleeplessness	MoodSwing	SuicidalThoughts	Anorxia	Dis
1	Usually	Seldom	Sometimes	Sometimes	YES	YES	NO	
2	Usually	Seldom	Usually	Sometimes	NO	YES	NO	
3	Sometimes	Most-Often	Sometimes	Sometimes	YES	NO	NO	
4	Usually	Seldom	Usually	Most-Often	YES	YES	YES	
5	Usually	Usually	Sometimes	Sometimes	NO	NO	NO	

Feature Engineering

In [11]:
```
df_encoded = pd.DataFrame()

# We use this to validate the values in a feature and
# scan for other values or typos.
def validate_unique_values(df, features, message=""):
    print(f"{message}:")

    try:
        for f in features:
            unique_values = df[f].unique()
            unique_values.sort()

            print(f"{f}: {unique_values}")
    except KeyError:
        print(f"ERROR - COLUMN '{f}' NOT FOUND")

    print()
```

Numerical Features

Set with discrete scale from 1 to 10: 'X From 10' (X)

1. Sexual Activity
2. Concentration
3. Optimisim

This data is debatably ordinal as it is ranked from 1 to 10. However, since we can infer an equal interval between the neighboring data points (the interval being 1) and there is approximately a natural 0 or absense of a value (which appears to be the minimum 1, though this scale could include 0 to which we had no instances), we can classify this measurement data as ratio. As such, we proceed by reformatting the values to their integer representation. We will return to scaling these features after examining their distributions in `04 - Exploratory Data Analysis`.

In [12]:
```
numerical_features = [
    'SexualActivity',
    'Concentration',
    'Optimisim'
```

```
]

validate_unique_values(df_mood, numerical_features, "1-10 Scale")
```

```
1-10 Scale:
SexualActivity: ['1 From 10' '2 From 10' '3 From 10' '4 From 10' '5 From 10' '6 From 10'
 '7 From 10' '8 From 10' '9 From 10']
Concentration: ['1 From 10' '2 From 10' '3 From 10' '4 From 10' '5 From 10' '6 From 10'
 '7 From 10' '8 From 10']
Optimisim: ['1 From 10' '2 From 10' '3 From 10' '4 From 10' '5 From 10' '6 From 10'
 '7 From 10' '8 From 10' '9 From 10']
```

In [13]:
```
df_encoded[numerical_features] = (
    df_mood[numerical_features]
    .apply(lambda col: col.str[0].astype('float'))
)

df_encoded[numerical_features].head()
```

Out[13]:

PersonNum	SexualActivity	Concentration	Optimisim
1	3.0	3.0	4.0
2	4.0	2.0	5.0
3	6.0	5.0	7.0
4	3.0	2.0	2.0
5	5.0	5.0	6.0

Ordinal Features

Set with four-point Likert scale: Seldom (0), Sometimes (1), Usually (2), Most-Often (3)

1. Sadness
2. Euphoria
3. Exhaustion
4. Sleeplessness

In [14]:
```
likert_features = [
    'Sadness',
    'Euphoria',
    'Exhaustion',
    'Sleeplessness'
]

validate_unique_values(df_mood, likert_features, "Likert Scale")
```

```
Likert Scale:
Sadness: ['Most-Often' 'Seldom' 'Sometimes' 'Usually']
Euphoria: ['Most-Often' 'Seldom' 'Sometimes' 'Usually']
Exhaustion: ['Most-Often' 'Seldom' 'Sometimes' 'Usually']
Sleeplessness: ['Most-Often' 'Seldom' 'Sometimes' 'Usually']
```

In [15]:
```
enc_likert = OrdinalEncoder(
    categories=[[
```

```
        'Seldom',
        'Sometimes',
        'Usually',
        'Most-Often'
]])

df_encoded[likert_features] = (
    df_mood[likert_features]
    .apply(lambda col: enc_likert.fit_transform(col.values.reshape(-1, 1)).flatten())
)

df_encoded[likert_features].head()
```

Out[15]:

PersonNum	Sadness	Euphoria	Exhaustion	Sleeplessness
1	2.0	0.0	1.0	1.0
2	2.0	0.0	2.0	1.0
3	1.0	3.0	1.0	1.0
4	2.0	0.0	2.0	3.0
5	2.0	2.0	1.0	1.0

Binary Features

Set with `YES` (1) and `NO` (0) values:

1. Mood Swing
2. Suicidal Thoughts
3. Anorxia
4. Disobedience
5. Justify Behavior
6. Aggressiveness
7. Move On
8. Nervous Breakdown
9. Admit Mistakes
10. Overthinking

In [16]:

```
binary_features = [
    'MoodSwing',
    'SuicidalThoughts',
    'Anorxia',
    'Disobedience',
    'JustifyBehavior',
    'Aggressiveness',
    'MoveOn',
    'NervousBreakdown',
    'AdmitMistakes',
    'Overthinking'
]

validate_unique_values(df_mood, binary_features, "Binary")
```

```
Binary:
MoodSwing: ['NO' 'YES']
SuicidalThoughts: ['NO' 'YES' 'YES ']
Anorxia: ['NO' 'YES']
Disobedience: ['NO' 'YES']
JustifyBehavior: ['NO' 'YES']
Aggressiveness: ['NO' 'YES']
MoveOn: ['NO' 'YES']
NervousBreakdown: ['NO' 'YES']
AdmitMistakes: ['NO' 'YES']
Overthinking: ['NO' 'YES']
```

In `SuicidalThoughts`, there is at least one instance containing `YES` (with a trailing space). We fix this before proceeding.

In [17]:
```
df_mood['SuicidalThoughts'] = df_mood['SuicidalThoughts'].str.strip()

validate_unique_values(df_mood, ['SuicidalThoughts'], "Suicidal Thoughts")
```

```
Suicidal Thoughts:
SuicidalThoughts: ['NO' 'YES']
```

In [18]:
```
enc_binary = OrdinalEncoder(categories=[['NO', 'YES']])

df_encoded[binary_features] = (
    df_mood[binary_features]
    .apply(lambda col: enc_binary.fit_transform(col.values.reshape(-1, 1)).flatten())
)

df_encoded[binary_features].head()
```

Out[18]:

PersonNum	MoodSwing	SuicidalThoughts	Anorxia	Disobedience	JustifyBehavior	Aggressiveness	Mov
1	1.0	1.0	0.0	0.0	1.0	0.0	
2	0.0	1.0	0.0	0.0	0.0	0.0	
3	1.0	0.0	0.0	0.0	1.0	1.0	
4	1.0	1.0	1.0	0.0	1.0	0.0	
5	0.0	0.0	0.0	0.0	0.0	0.0	

Nominal Target Class

The target class is `Diagnosis`, which is a classification of the mood disoder.

In [19]:
```
target_name = 'Diagnosis'

validate_unique_values(df_mood, [target_name], "Target")
```

```
Target:
Diagnosis: ['Bipolar Type-1' 'Bipolar Type-2' 'Depression' 'Normal']
```

In [20]:
```
df_encoded['Diagnosis'] = df_mood['Diagnosis'].astype(str)
```

04 - Exploratory Data Analysis

Processed Features' Data Types

In [21]:
```
for column in df_encoded.columns:
    print(f'{column}: {df_encoded[column].dtype}')
```

```
SexualActivity: float64
Concentration: float64
Optimisim: float64
Sadness: float64
Euphoria: float64
Exhaustion: float64
Sleeplessness: float64
MoodSwing: float64
SuicidalThoughts: float64
Anorxia: float64
Disobedience: float64
JustifyBehavior: float64
Aggressiveness: float64
MoveOn: float64
NervousBreakdown: float64
AdmitMistakes: float64
Overthinking: float64
Diagnosis: object
```

Having processed the features and target class, all data types except for the response variable are now `float64`. The range of each variable can be found both in the processing section and in the summary statistics below.

Summary Statistics

In [22]:
```
display(
    df_encoded[numerical_features + likert_features]
    .describe()
    .style
    .set_table_attributes("style='display:inline'")
    .set_caption('Nonbinary Predictors')
)
```

Nonbinary Predictors

	SexualActivity	Concentration	Optimisim	Sadness	Euphoria	Exhaustion	Sleeplessness
count	240.000000	240.000000	240.000000	240.000000	240.000000	240.000000	240.000000
mean	4.741667	4.250000	4.466667	1.550000	0.933333	1.633333	1.458333
std	2.006249	1.793760	1.987127	0.922522	0.921463	1.018107	0.975824
min	1.000000	1.000000	1.000000	0.000000	0.000000	0.000000	0.000000
25%	3.000000	3.000000	3.000000	1.000000	0.000000	1.000000	1.000000
50%	5.000000	4.000000	4.000000	2.000000	1.000000	2.000000	1.000000
75%	6.000000	5.000000	6.000000	2.000000	1.000000	2.250000	2.000000
max	9.000000	8.000000	9.000000	3.000000	3.000000	3.000000	3.000000

In [23]:
```python
def describe_binary(df):
    stats = {
        'Count': [],
        'Count True': [],
        'Count False': [],
        'Mean': []
    }

    for column in df.columns:
        counts = df[column].value_counts()
        stats['Count'].append(counts.sum())
        stats['Count True'].append(counts.get(1, 0))
        stats['Count False'].append(counts.get(0, 0))
        stats['Mean'].append(counts.get(1, 0) / counts.sum() if counts.sum() > 0 else 0)

    return pd.DataFrame(stats, index=df.columns).transpose()

display(
    describe_binary(df_encoded[binary_features])
    .style
    .set_table_attributes("style='display:inline'")
    .set_caption('Binary Predictors')
)
```

Binary Predictors

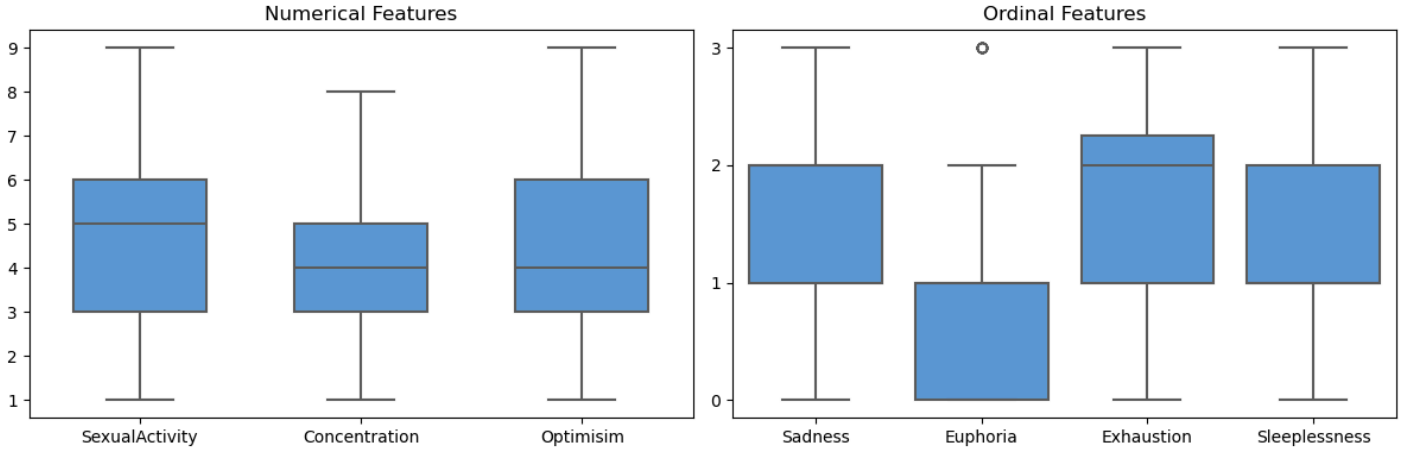
	MoodSwing	SuicidalThoughts	Anorxia	Disobedience	JustifyBehavior	Aggressiveness	MoveO
Count	240.000000	240.000000	240.000000	240.000000	240.000000	240.000000	240.00000
Count True	114.000000	114.000000	92.000000	94.000000	114.000000	116.000000	100.00000
Count False	126.000000	126.000000	148.000000	146.000000	126.000000	124.000000	140.00000
Mean	0.475000	0.475000	0.383333	0.391667	0.475000	0.483333	0.41666

Data Distributions & Outlier Analysis

To scan for outliers, we begin by constructing boxplots of our non-binary data:

In [24]:
```
# Prepare data for visualization by converting to
# long format.
df_melt_numerical = (
    df_encoded[numerical_features]
    .melt(var_name='Numerical', value_name='Scale')
)

df_melt_likert = (
    df_encoded[likert_features]
    .melt(var_name='Likert', value_name='Frequency')
)
```

In [25]:
```
colors = ['#E65C7E', '#469AEA']
sns.set_palette(sns.color_palette(colors))

fig, axes = plt.subplots(1, 2, figsize=(12, 4))

ax11 = sns.boxplot(
    data=df_melt_numerical,
    x='Numerical',
    y='Scale',
    width=0.6,
    linewidth=1.5,
    ax=axes[0],
    color=colors[1]
)
axes[0].set_title('Numerical Features')
axes[0].set_xlabel('')
axes[0].set_ylabel('')

ax21 = sns.boxplot(
    data=df_melt_likert,
    x='Likert',
    y='Frequency',
    linewidth=1.5,
    ax=axes[1],
    color=colors[1]
)
axes[1].set_title('Ordinal Features')
axes[1].set_xlabel('')
axes[1].set_ylabel('')
axes[1].set_yticks([0, 1, 2, 3])

plt.tight_layout()
plt.show()
```



Euphoria is the only feature for which an outlier (equal to 3, or "Sometimes") was detected via IQR. We investigate this value further.

In [26]:
```
cond = df_encoded['Euphoria'] == 3

total_obs = df_encoded.shape[0]
favor_obs = df_encoded[cond].shape[0]

ratio_obs = favor_obs / total_obs

print(f"""
Number of Total Observations: {total_obs}
Number of Observations where Euphoria = 3: {favor_obs}
Ratio of Observations where Euphoria = 3: {ratio_obs}
""")
```

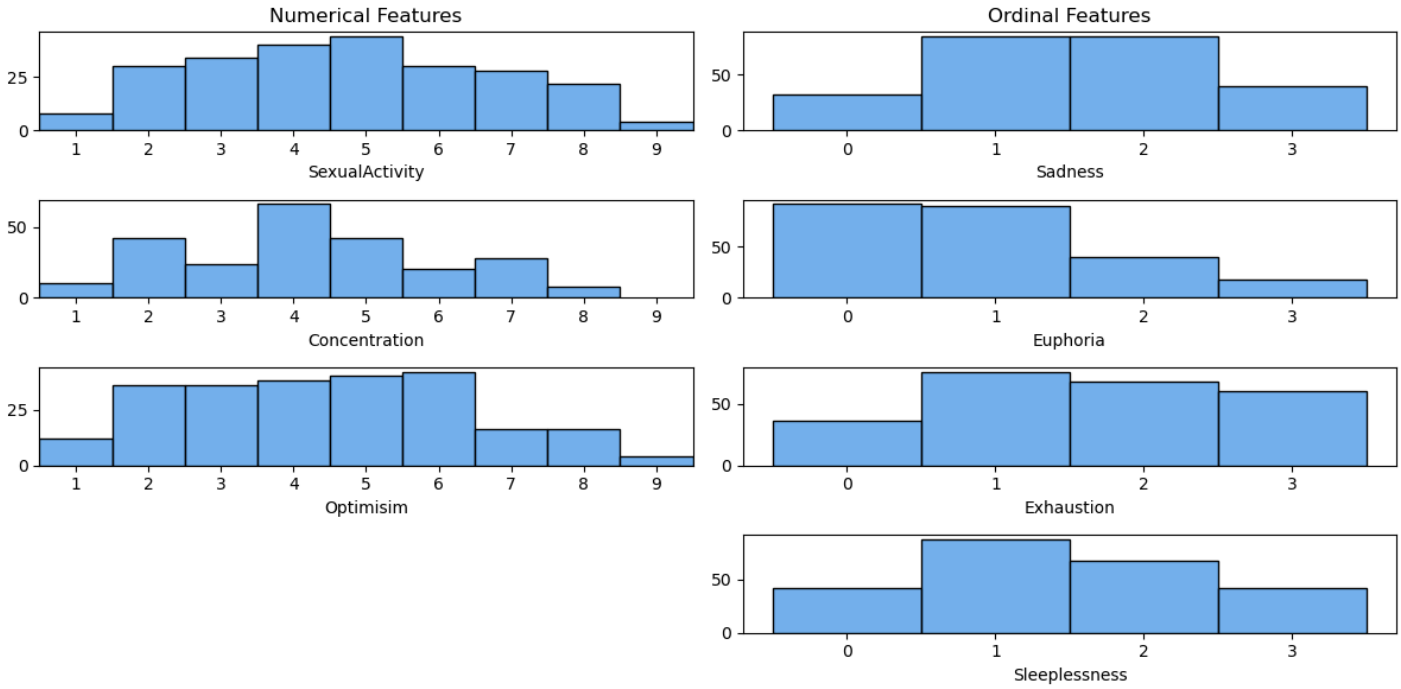
```
Number of Total Observations: 240
Number of Observations where Euphoria = 3: 18
Ratio of Observations where Euphoria = 3: 0.075
```

Proportionally, the number of observations in which Euphoria is 3 is low. However, due to there being 18 instances of this value and that this value is achievable on the given scale, we do not consider these values to be outliers. Thus, no outliers were detected in our non-binary data.

Now, consider the histograms of our data with regard to the distribution of the data:

In [27]:
```
fig, axes = plt.subplots(4, 2, figsize=(12, 6))
axes = axes.flatten()  # for better iteration over the grid

for idx, col in enumerate(numerical_features):
    ax_pos = idx * 2

    sns.histplot(
        x=df_encoded[col],
        discrete=True,
        color=colors[1],
        ax=axes[ax_pos]
    )
    if idx == 0:
        axes[ax_pos].set_title('Numerical Features')
    axes[ax_pos].set_xlabel(col)
    axes[ax_pos].set_ylabel('')
    axes[ax_pos].set_xlim(0.5, 9.5)

for idx, col in enumerate(likert_features):
    ax_pos = idx * 2 + 1

    sns.histplot(
        x=df_encoded[col],
        discrete=True,
        color=colors[1],
        ax=axes[ax_pos]
    )
    if idx == 0:
        axes[ax_pos].set_title('Ordinal Features')
    axes[ax_pos].set_xlabel(col)
    axes[ax_pos].set_ylabel('')
    axes[ax_pos].set_xticks([0, 1, 2, 3])
```

```
# Remove the empty subplot
fig.delaxes(axes[6])

plt.tight_layout()
plt.show()
```



By visual inspection, we find that `SexualActivity`, `Sadness`, `Exhaustion`, and `Sleeplessness` can all be approximated with a normal distribution. `Concentration` is somewhat normal with chaotic tails, `Optimism` could be is somewhat normal with left-skewness, and `Euphoria` is heavily right-skewed.

To scan for outliers and observe the distributions in our binary data, we use bar charts to show the frequency of "YES" vs. "NO" values:

In [32]:
```
# Prepare data for visualization
# by calculating value counts.
binary_counts_per_column = (
    df_encoded[binary_features]
    .apply(pd.Series.value_counts)
)

# Reset index to make the DataFrame
# easier to plot
binary_counts_per_column = (
    binary_counts_per_column
    .T
    .reset_index()
    .melt(id_vars=['index'], value_vars=[0, 1])
)

binary_counts_per_column.columns = [
    'Feature',
    'Value',
    'Count'
]
```

In [29]:
```
fig, axes = plt.subplots(figsize=(12, 6))

ax3 = sns.barplot(
    data=binary_counts_per_column,
```

```
    x='Count',
    y='Feature',
    hue='Value',
    width=0.6
)

# Create borders to the bars
for bar in ax3.patches:
    bar.set_edgecolor('#222222')
    bar.set_linewidth(1)

plt.title('Binary Features')
plt.xlabel('')
plt.ylabel('')
plt.legend(title='Value', labels=['NO', 'YES'])

plt.tight_layout()
plt.show()
```

We find no instances of outliers in our binary features.

Lastly, we turn to visualizing the distribution of the target variable, `diagnosis`:

In [31]:
```
target_counts_per_column = (
    df_encoded[['Diagnosis']]
    .apply(pd.Series.value_counts)
)

# Reset index to make the DataFrame
# easier to plot
target_counts_per_column = (
    target_counts_per_column
    .T
    .reset_index()
    .melt(id_vars=['index'])
    .sort_values('Diagnosis')
)

target_counts_per_column.columns = [
    'Target',
```

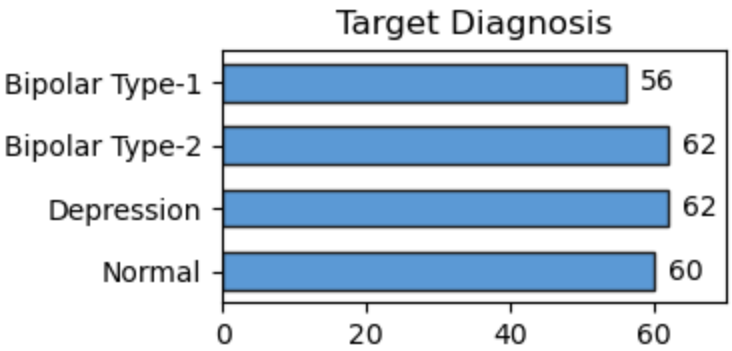
```
    'Diagnosis',
    'Count'
]

target_counts_per_column.head()
```

Out[31]:

	Target	Diagnosis	Count
3	Diagnosis	Bipolar Type-1	56
0	Diagnosis	Bipolar Type-2	62
1	Diagnosis	Depression	62
2	Diagnosis	Normal	60

In [33]:

```
fig, axes = plt.subplots(figsize=(4, 2))

sns.barplot(
    data=target_counts_per_column,
    x='Count',
    y='Diagnosis',
    width=0.6,
    color=colors[1],
    edgecolor='#222222',
    linewidth=1,
    ax=axes
)
plt.title('Target Diagnosis')
plt.xlabel('')
plt.ylabel('')
plt.xlim(0, 70)

# Add bar counts to the figure.
for container in axes.containers:
    axes.bar_label(container, padding=5)

plt.tight_layout()
plt.show()
```



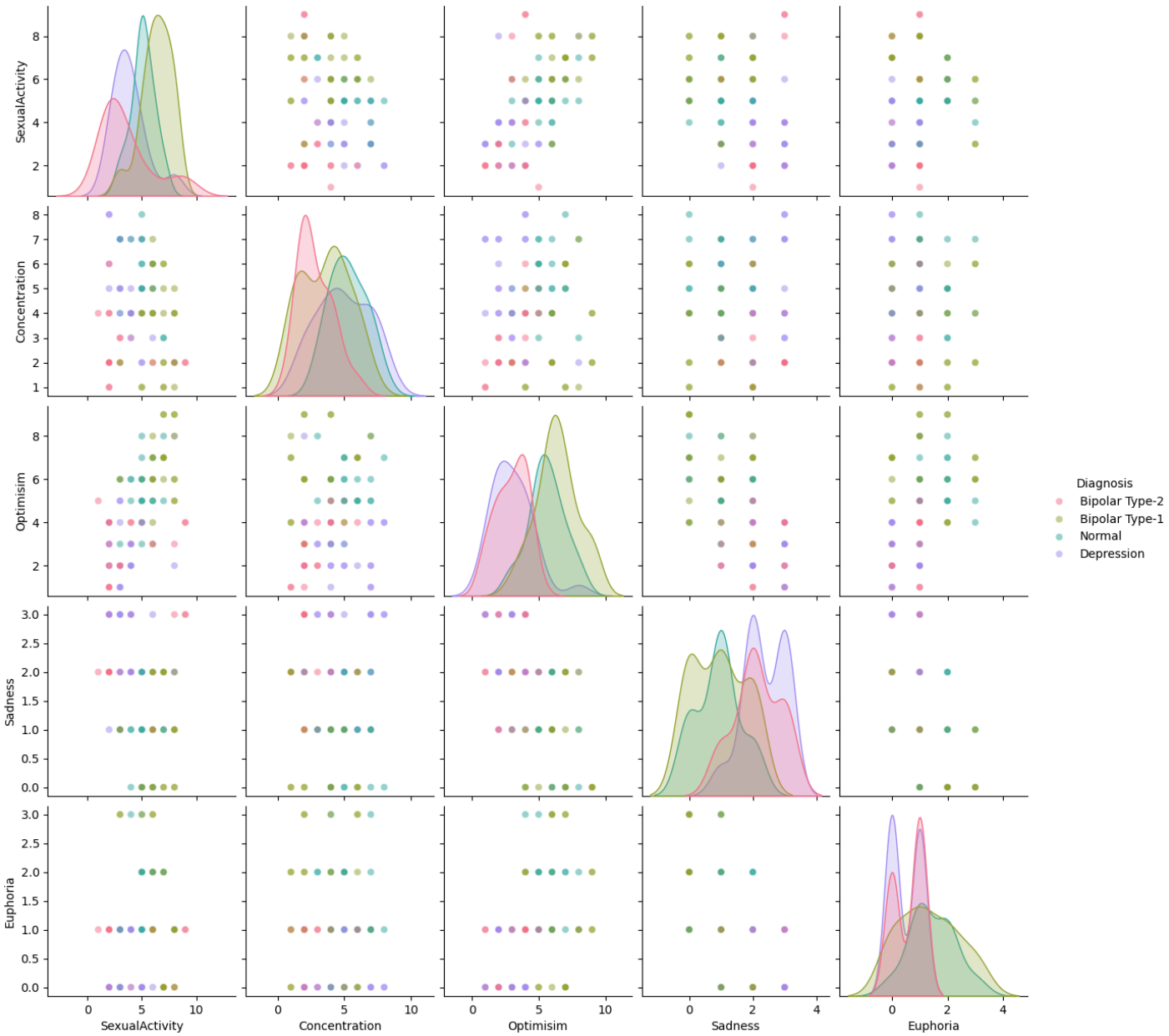
We find that the frequency of all of our diagnoses is relatively the same.

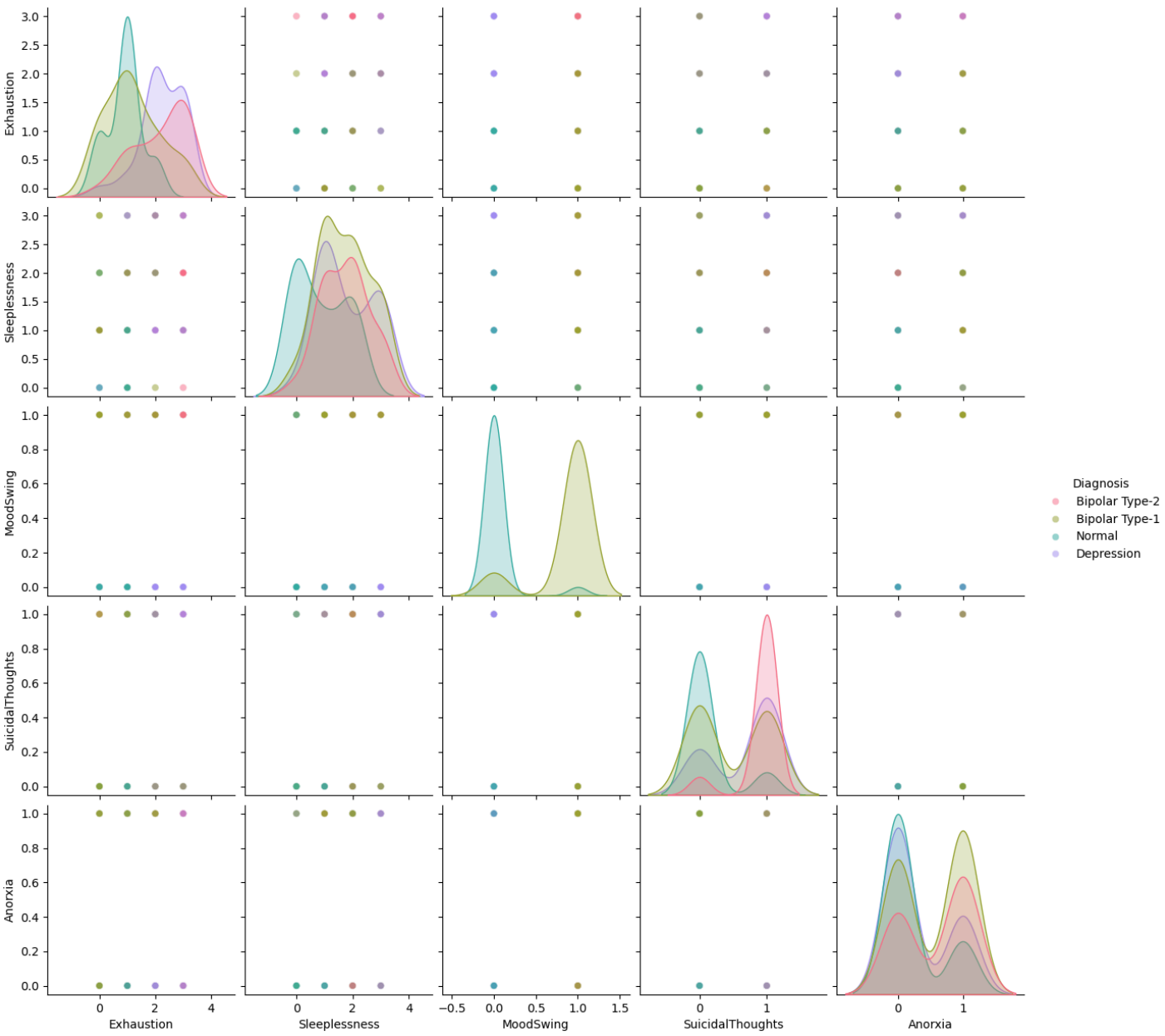
Pair Plots

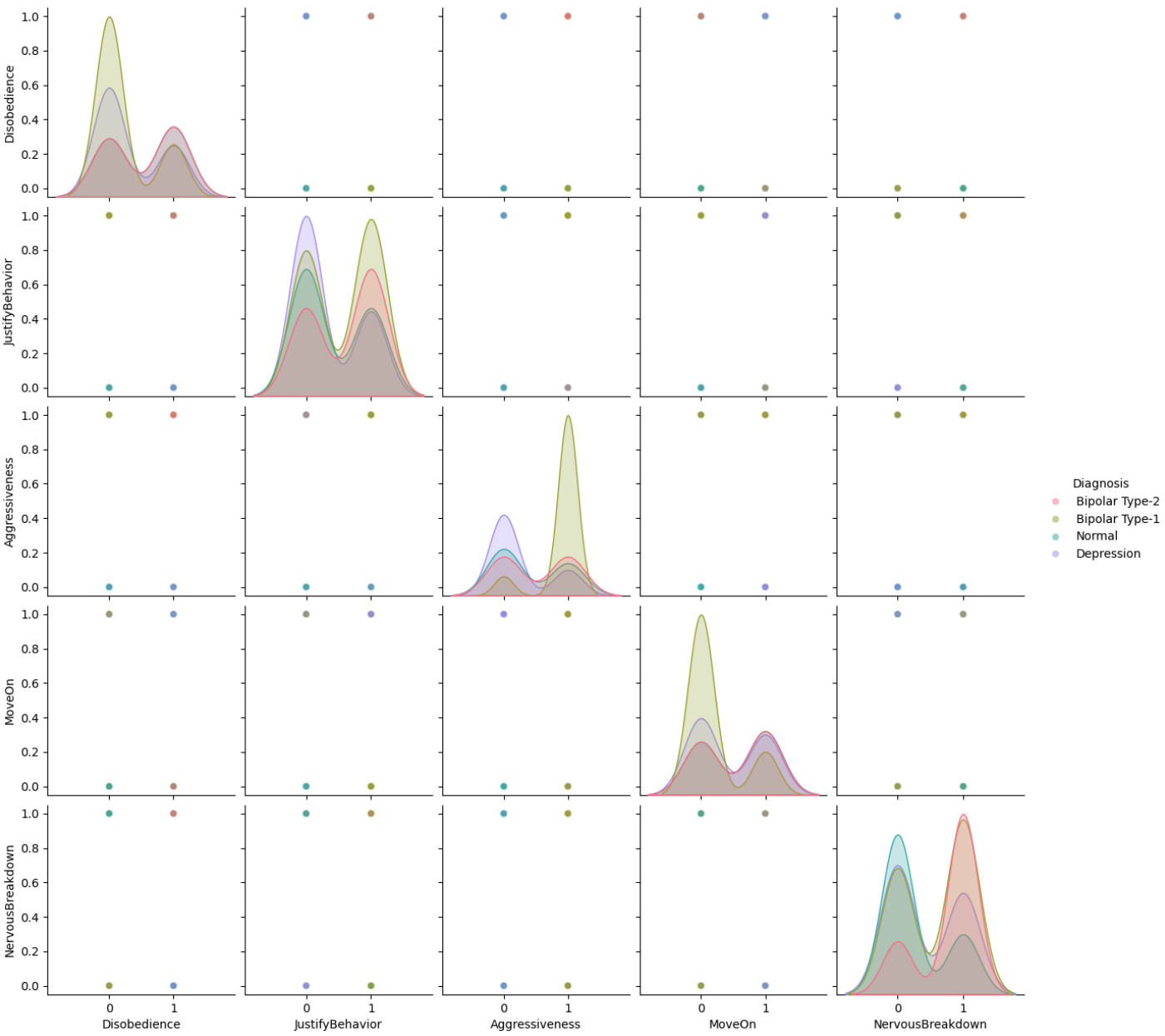
In [37]:

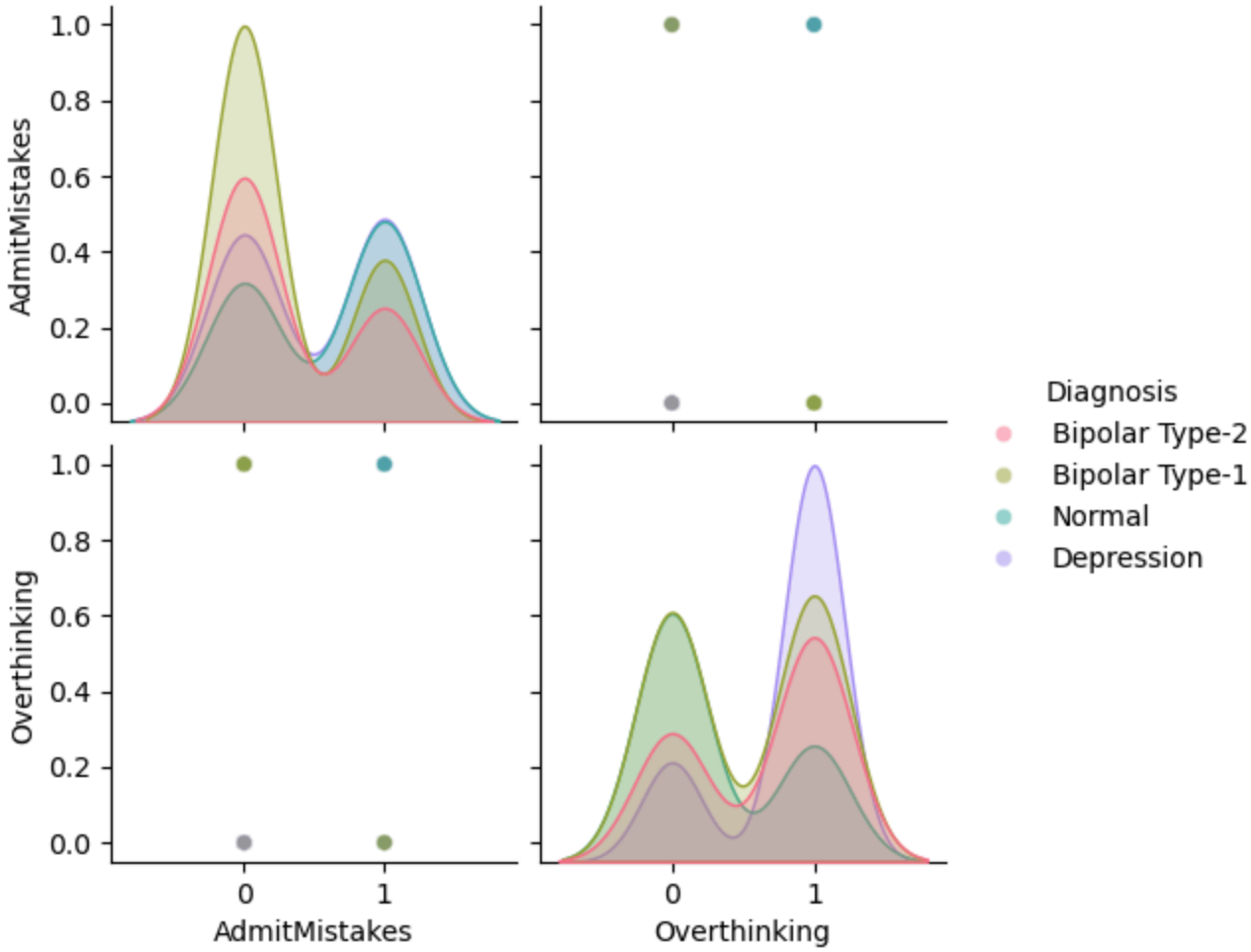
```
# Define function to create pair plots in segments
def create_pair_plots(data, target_column, chunk_size=5):
    predictors = data.columns[:-1]
    for i in range(0, len(predictors), chunk_size):
```

```
        subset = list(predictors[i:i + chunk_size]) + [target_column]

        sns.pairplot(
            data,
            vars=subset[:-1],
            hue=target_column,
            plot_kws={'alpha': 0.5},
            height=2.5
        )
        plt.show()

df_sampled = df_encoded.sample(n=100, random_state=6043)

create_pair_plots(df_sampled, 'Diagnosis')
```




Exhaustion
Sleeplessness
MoodSwing
SuicidalThoughts
Anorxia
Diagnosis
Bipolar Type-2
Bipolar Type-1
Normal
Depression

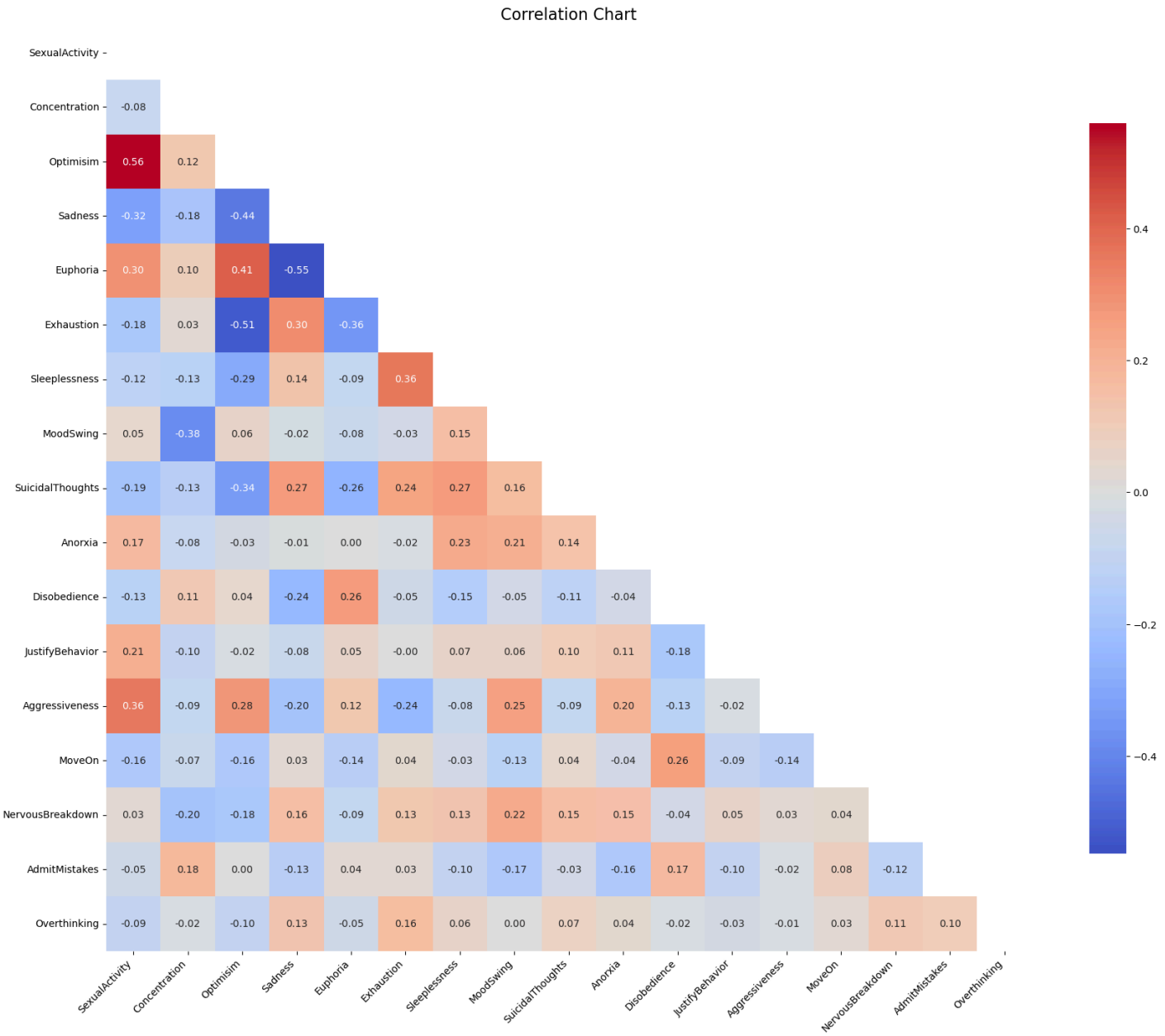

Disobedience
JustifyBehavior
Aggressiveness
MoveOn
NervousBreakdown
0.0
0.2
0.4
0.6
0.8
1.0
0
1
Diagnosis
Bipolar Type-2
Bipolar Type-1
Normal
Depression



Correlation Chart

We now turn to observing relationships between the features by considering their pariwise correlations. Note that correlation measures the strength of the fit as a linear relationship, so pairwise features not linearly related might have a smaller relationship.

In [38]:
```
corr = df_encoded.drop(columns="Diagnosis").corr()

# Getting the Upper Triangle of the correlation matrix
plt.figure(figsize=(18, 15))

matrix = np.triu(corr)

ax4 = sns.heatmap(
    corr,
    mask=matrix,
    center=0,
    square=True,
    cmap='coolwarm',
    annot=True,
    annot_kws={"size": 10},
    fmt=".2f",
    cbar_kws={'shrink': 0.75}
)
plt.title('Correlation Chart', fontsize=16)
plt.xticks(rotation=45, ha='right')

plt.tight_layout()
plt.show()
```



Some of the most notable correlations include the following:

Postive

1. Optimism & Sexual Activity (0.56)
2. Optimism & Euphoria (0.41)
3. Sleeplessness & Exhaustion (0.36)
4. Sexual Activity & Aggressiveness (0.36)

Negative

1. Sadness & Euphoria (-0.55)
2. Exhaustion & Optimism (-0.51)
3. Sadness & Optimism (-0.44)
4. Mood Swing & Concentration (-0.38)

Generally speaking, these findings hold true with our expectations.

Scaling the Data

Having observed the distributions on our data and found no outliers, we proceed to scaling our data. For numeric data, we utilize `StandardScaler` for normal distributions and `MinMaxScaler` for non-normal distributions. As our Likert scale ordinal data cannot be assumed to have equal intervals between the categories, we maintain the integer ordering without any scaling. Additionally, our binary data is already encoded.

Note that scaling is necessary for KNN, but not necessary for tree-based learning models. However, scaling will not negatively influence the performance of tree-based learning models, so our scaled features will persist through the remainder of the project.

In [43]:
```
# The following features are approximately normal.
normal_scaler = StandardScaler()

try:
    df_encoded['SexualActivity'] = (
        normal_scaler
        .fit_transform(df_encoded[['SexualActivity']])
    )

    df_encoded['Optimisim'] = (
        normal_scaler
        .fit_transform(df_encoded[['Optimisim']])
    )

except KeyError as e:
    print(f"ERROR - COLUMN '{e.args[0]}' NOT FOUND")

# The following features are not normal.
nonnormal_scaler = MinMaxScaler()
try:
    df_encoded['Concentration'] = (
        nonnormal_scaler
        .fit_transform(df_encoded[['Concentration']])
    )

except KeyError as e:
    print(f"ERROR - COLUMN '{e.args[0]}' NOT FOUND")

df_encoded[numerical_features].head()
```

Out[43]:

PersonNum	SexualActivity	Concentration	Optimisim
1	-0.869935	0.285714	-0.235336
2	-0.370451	0.142857	0.268955
3	0.628518	0.571429	1.277536
4	-0.869935	0.142857	-1.243917
5	0.129033	0.571429	0.773246

05 - Model Building

Initial Modeling

Having processed and provided a rudimentary analysis of our data, we now focus on creating learning models in an attempt to build a classifier for future mood disorder predictions.

In [46]:
```
target = df_encoded[['Diagnosis']]
features = df_encoded.drop(columns=['Diagnosis'])

# There is no need for column selector since all of
# the columns have already been encoded numerically.

try:
    x_train, x_test, y_train, y_test = train_test_split(
        features,
        target,
        test_size=0.25,
        random_state=6043
    )

except Exception as e: print(f"ERROR - TRAIN-TEST SPLIT: {e}")

# There is no need to utilize the preprocessing methods
# since this was accomplished in sections 03 and 04.
```

In [47]:
```
models = {
    'KNN': KNeighborsClassifier(),
    'DecisionTree': DecisionTreeClassifier(random_state=6043),
    'RandomForest': RandomForestClassifier(random_state=6043),
    'ExtraTree': ExtraTreesClassifier(random_state=6043)
}

# Training #1: Using default hyperparameter values
try:
    for model_name, model in models.items():
        model.fit(x_train, np.ravel(y_train))

        y_pred = model.predict(x_test)
        accuracy = accuracy_score(y_test, y_pred)

        print(f"Accuracy of {model_name}: {accuracy:.3f}")

except Exception as e:
    print(f"ERROR - WITH {model_name}: {e}")
```

```
Accuracy of KNN: 0.800
Accuracy of DecisionTree: 0.967
Accuracy of RandomForest: 0.967
Accuracy of ExtraTree: 0.967
```

Out of all the models selected, the tree-based models have the highest accuracy based on the default hyperparameter values. We fine-tune the hyperparameters to potentially improve performance.

Hyperparameter Tuning

In [48]:
```
# Use this for fitting hyperparameters to a model
def fit_hypers(model, params, x_train, y_train):
    try:
        model_grid_param = params

        model_grid_search = GridSearchCV(
            model,
```

```
            model_grid_param,
            cv=5,
            scoring='f1_micro',
            n_jobs=-1
        )

        fitted_model = model_grid_search.fit(
            x_train,
            np.ravel(y_train)
        )

        return fitted_model

    except Exception as e:
        print(f"ERROR - {e}")
        return None
```

KNN

In [49]:
```
knn_params = {'n_neighbors': list(range(2,13))}

knn_fitted_model = fit_hypers(
    models['KNN'],
    knn_params,
    x_train,
    y_train
)

print(knn_fitted_model.best_params_)
```

```
{'n_neighbors': 2}
```

Decision Tree

In [50]:
```
dt_params = {
    'criterion': ['gini', 'entropy'],
    'max_depth': [None, 10, 20, 30],
    'min_samples_leaf': [1, 2, 4, 6, 8],
    'min_samples_split': [2, 5, 10]
}

dt_fitted_model = fit_hypers(
    models['DecisionTree'],
    dt_params,
    x_train,
    y_train
)

print(dt_fitted_model.best_params_)
```

```
{'criterion': 'gini', 'max_depth': None, 'min_samples_leaf': 1, 'min_samples_split': 2}
```

Random Forest

In [51]:
```
rf_params = {
    'n_estimators': [50, 100, 150],
    'max_depth': [None, 10, 20, 30],
    'min_samples_leaf': [1, 2, 4, 6, 8],
    'min_samples_split': [2, 5, 10]
}
```

```
rf_fitted_model = fit_hypers(
    models['RandomForest'],
    rf_params,
    x_train,
    y_train
)

print(rf_fitted_model.best_params_)
```

```
{'max_depth': None, 'min_samples_leaf': 1, 'min_samples_split': 2, 'n_estimators': 50}
```

Extra Tree

In [52]:
```
et_params = {
    'n_estimators': [50, 100, 150],
    'max_depth': [None, 10, 20, 30],
    'min_samples_leaf': [1, 2, 4, 6, 8],
    'min_samples_split': [2, 5, 10]
}

et_fitted_model = fit_hypers(
    models['ExtraTree'],
    et_params,
    x_train,
    y_train
)

print(et_fitted_model.best_params_)
```

```
{'max_depth': None, 'min_samples_leaf': 1, 'min_samples_split': 2, 'n_estimators': 50}
```

06 - Model Evaluation

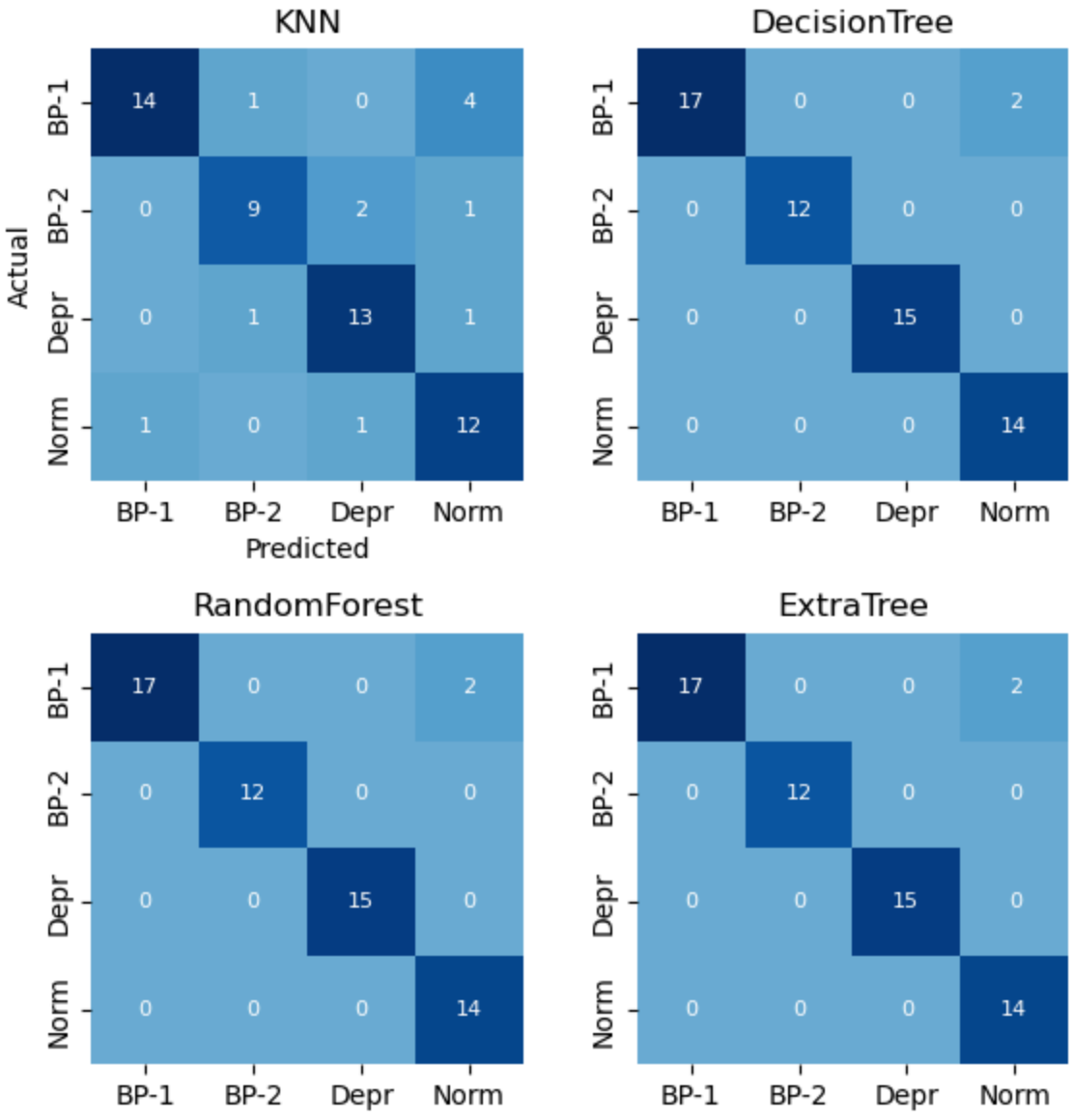
Finally, we evaluate the model based on the following performance metrics:

- Confusion Matrix
- Performance Statistics:
  - Accuracy
  - Precision
  - Recall (Sensitivity)
  - Specificity
  - F1 Score
- K-Fold Cross Validation

Confusion Matrix

In [53]:
```
fig, axes = plt.subplots(2, 2, figsize=(6, 6))

# Order checked with:
# y_test.value_counts()
display_labels = ['BP-1', 'BP-2', 'Depr', 'Norm']

idx = 0
for model_name, model in models.items():
    ax_pos = idx // 2, idx % 2
```

```
    y_pred = model.predict(x_test)
    cm = confusion_matrix(y_test, y_pred)

    sns.heatmap(
        cm,
        ax=axes[ax_pos[0], ax_pos[1]],
        xticklabels=display_labels,
        yticklabels=display_labels,
        center=0,
        square=True,
        cmap='Blues',
        annot=True,
        annot_kws={"size": 8},
        fmt='d',
        cbar=False
    )
    axes[ax_pos[0], ax_pos[1]].set_title(model_name)
    if idx == 0:
        axes[ax_pos[0], ax_pos[1]].set_xlabel('Predicted')
        axes[ax_pos[0], ax_pos[1]].set_ylabel('Actual')

    idx += 1

plt.tight_layout()
plt.show()
```



Performance Metrics

In [54]:
```python
def model_composite_eval(models, x_train, x_test, y_test):
    df_results = pd.DataFrame(columns=[
        'Model',
        'Accuracy',
        'Precision',
        'Recall',
        'Specificity',
        'F1 Score'
    ])

    for model_name, model in models.items():
        try:
            y_pred = model.predict(x_test)

            # (TP + TN) / (TP + TN + FP + FN)
            accuracy = accuracy_score(y_test, y_pred)

            # Since the target classes are balanced, we use macro
            # averaging to evaluate the performance on each class
            # equally and it provides insights into how well the
            # model performs across all classes independently

            # (1/n) * SUM(TP_i / (TP_i + FP_i))
            precision = precision_score(y_test, y_pred, average='macro')

            # (1/n) * SUM(TP_i / (TP_i + FN_i))
            recall = recall_score(y_test, y_pred, average='macro')

            # TN / (TN + FP)
            specificity = specificity_score(y_test, y_pred, average='macro')

            # 2 * (Precision * Recall) / (Precision + Recall)
            f1_score = model.best_score_

            results = [
                model_name,
                accuracy,
                precision,
                recall,
                specificity,
                f1_score
            ]

            # Append results to the results Dataframe.
            df_results.loc[len(df_results)] = results

        except Exception as e:
            print(f"ERROR - WITH {model_name}: {e}")

    return df_results

models_fitted = {
    'KNN': knn_fitted_model,
    'DecisionTree': dt_fitted_model,
    'RandomForest': rf_fitted_model,
    'ExtraTree': et_fitted_model
}
```

```
df_results = model_composite_eval(
    models_fitted,
    x_train,
    x_test,
    y_test
)

df_results
```

Out[54]:

	Model	Accuracy	Precision	Recall	Specificity	F1 Score
0	KNN	0.950000	0.960565	0.946429	0.982249	0.844444
1	DecisionTree	0.966667	0.968750	0.973684	0.989130	0.922222
2	RandomForest	0.966667	0.968750	0.973684	0.989130	0.950000
3	ExtraTree	0.966667	0.968750	0.973684	0.989130	0.961111

K-Fold Cross-Validation

In [55]:

```
def get_cv_score(model, x_train, y_train, kfold, metric):
    try:
        cv_scores = cross_val_score(
                model,
                x_train,
                np.ravel(y_train),
                cv=kfold,
                scoring=metric,
                n_jobs=-1
            )

        return np.mean(cv_scores)

    except Exception as e:
        print(f"ERROR - CROSS-VALIDATION: {e}")
        return

def model_kfcv_eval(models, x_train, y_train, x_test, y_test):
    # Removed `Specificity` as not supported by SciKit-Learn
    df_kf_results = pd.DataFrame(columns=[
        'Model',
        'Accuracy',
    ])

    for model_name, model in models.items():
        kfold = KFold(
            n_splits=3,
            shuffle=True,
            random_state=6043
        )

        accuracy_kf = get_cv_score(
            model,
            x_train,
            y_train,
            kfold,
            'accuracy'
        )
```

```
        results = [model_name, accuracy_kf]
        # Append results to the results Dataframe.
        df_kf_results.loc[len(df_kf_results)] = results

    return df_kf_results

df_kf_results = model_kfcv_eval(
    models_fitted,
    x_train,
    y_train,
    x_test,
    y_test
)

df_kf_results
```

Out[55]:

	Model	Accuracy
0	KNN	0.805556
1	DecisionTree	0.911111
2	RandomForest	0.911111
3	ExtraTree	0.961111

Using 3-fold cross-validation, the most accurate model identified was ExtraTree, optimized with the following hyperparameters: `max_depth=None`, `min_samples_leaf=1`, `min_samples_split=2`, and `n_estimators=50`. This model achieved an impressive performance, with an accuracy of 96.67%, a precision of 96.88%, a recall of 97.34%, and an F1 score of 96.11%. Based on our analysis, we believe this model offers a robust approach to identifying mood disorders in patients. By improving the accuracy of these diagnoses, we can facilitate timely and effective treatment, ultimately making a meaningful impact on patients' lives.