Machine Learning-Based Analysis of Radio Coverage Information and Network Scenarios for LTE Optimization

In [4]:
```
# Standard Libraries
import numpy as np
import pandas as pd
import random
import warnings
import matplotlib.pyplot as plt
import seaborn as sns

# Machine Learning & Preprocessing
from sklearn.preprocessing import StandardScaler, LabelEncoder, OneHotEncoder, label_binarize
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from xgboost import XGBClassifier
from sklearn.cluster import KMeans, DBSCAN
from sklearn.neighbors import NearestNeighbors
from sklearn.multiclass import OneVsRestClassifier

# Evaluation Metrics
from sklearn.metrics import (
    accuracy_score, precision_score, recall_score, f1_score,
    classification_report, confusion_matrix, silhouette_score,
    roc_curve, auc, precision_recall_curve, roc_auc_score,
    ConfusionMatrixDisplay, RocCurveDisplay
)

# Hyperparameter Optimization
from sklearn_genetic import GASearchCV
from sklearn_genetic.space import Integer, Categorical
from sklearn.exceptions import ConvergenceWarning

# Genetic Algorithm for clustering optimization
from deap import base, creator, tools, algorithms

# Utilities
from kneed import KneeLocator
from scipy import stats

# Suppress warnings
warnings.simplefilter("ignore", ConvergenceWarning)
```

TASK 1

DATA PREPROCESSING AND EXPLORATION

In [6]:
```
# Load dataset
file_path = "4G - Passive measurements.csv"  # This file is placed in the working directory
df = pd.read_csv(file_path)
```

In [7]:
```
# Shows the total number of rows and columns within the dataset
df.shape
```

Out[7]: (527540, 27)

In [8]:
```
# Display basic information about the dataset
print("Dataset Info:")
df.info()
```

```
Dataset Info:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 527540 entries, 0 to 527539
Data columns (total 27 columns):
 #   Column               Non-Null Count   Dtype
---  ------               --------------   -----
 0   Unnamed: 0           527540 non-null  int64
 1   Date                 527540 non-null  object
 2   Time                 527540 non-null  object
 3   UTC                  457201 non-null  float64
 4   Latitude             527540 non-null  float64
 5   Longitude            527540 non-null  float64
 6   Altitude             457201 non-null  float64
 7   Speed                457201 non-null  float64
 8   EARFCN               527540 non-null  int64
 9   Frequency            527540 non-null  float64
 10  PCI                  527540 non-null  int64
 11  MNC                  527540 non-null  object
 12  CellIdentity         527540 non-null  int64
 13  eNodeB.ID            527540 non-null  int64
 14  Power                505826 non-null  float64
 15  SINR                 505826 non-null  float64
 16  RSRP                 527540 non-null  float64
 17  RSRQ                 527540 non-null  float64
 18  scenario             527540 non-null  object
 19  cellLongitude        527540 non-null  float64
 20  cellLatitude         527540 non-null  float64
 21  cellPosErrorLambda1  519632 non-null  float64
 22  cellPosErrorLambda2  527540 non-null  float64
 23  n_CellIdentities     527540 non-null  int64
 24  distance             527540 non-null  float64
 25  Band                 527540 non-null  int64
 26  campaign             527540 non-null  object
dtypes: float64(15), int64(7), object(5)
memory usage: 108.7+ MB
```

In [9]:
```
# Display first few rows to understand the structure
print("\nFirst 5 rows:")
print(df.tail())
```

```
First 5 rows:
        Unnamed: 0        Date          Time           UTC   Latitude  \
527535       43501  13.12.2020  11:54:37.247  1.610535e+09  41.871578
527536       43521  13.12.2020  11:54:37.247  1.610535e+09  41.871578
527537       43611  13.12.2020  11:54:37.439  1.610535e+09  41.871578
527538       43621  13.12.2020  11:54:37.439  1.610535e+09  41.871578
527539       43631  13.12.2020  11:54:37.439  1.610535e+09  41.871578

        Longitude  Altitude  Speed  EARFCN  Frequency  ...   RSRQ scenario  \
527535  12.463104     97.87   0.61    1350     1820.0  ... -19.26       IS
527536  12.463104     97.87   0.61    1350     1820.0  ... -17.85       IS
527537  12.463104     97.87   0.61    3175     2662.5  ... -19.57       IS
527538  12.463104     97.87   0.61    3175     2662.5  ... -11.87       IS
527539  12.463104     97.87   0.61    3175     2662.5  ... -16.22       IS

        cellLongitude  cellLatitude  cellPosErrorLambda1  cellPosErrorLambda2  \
527535      12.463643     41.871855            19.560003            19.560003
527536      12.461764     41.871894            11.785002            11.785002
527537      12.461764     41.871894            20.130005            20.130005
527538      12.461764     41.871894            20.130005            20.130005
527539      12.463643     41.871855            18.040005            18.040005

        n_CellIdentities    distance Band           campaign
527535                 5   54.286957    3  campaign_118_IS_4G
527536                 8  116.513722    3  campaign_118_IS_4G
527537                 8  116.513722    7  campaign_118_IS_4G
527538                 8  116.513722    7  campaign_118_IS_4G
527539                 5   54.286957    7  campaign_118_IS_4G

[5 rows x 27 columns]
```

In [10]:
```
# Removes the first column, which is an identifier that is irrelevant
df = df.drop(columns=["Unnamed: 0"], errors="ignore")
```

In [11]:
```
# Check for duplicates
duplicate_rows = df.duplicated().sum()
print(f"\nNumber of duplicate rows: {duplicate_rows}")
```

```
Number of duplicate rows: 0
```

In [12]:
```
# Check for missing values
print("\nMissing Values:")
print(df.isnull().sum())
```

```
Missing Values:
Date                       0
Time                       0
UTC                    70339
Latitude                   0
Longitude                  0
Altitude               70339
Speed                  70339
EARFCN                     0
Frequency                  0
PCI                        0
MNC                        0
CellIdentity               0
eNodeB.ID                  0
Power                  21714
SINR                   21714
RSRP                       0
RSRQ                       0
scenario                   0
cellLongitude              0
cellLatitude               0
cellPosErrorLambda1     7908
cellPosErrorLambda2        0
n_CellIdentities           0
distance                   0
Band                       0
campaign                   0
dtype: int64
```

In [13]:

```python
# Calculate missing values and total rows
missing_values = df.isnull().sum()
total_rows = len(df)

# Calculate missing percentages
missing_percent = (missing_values / total_rows) * 100
non_missing_percent = 100 - missing_percent

# Create a DataFrame for plotting
missing_df = pd.DataFrame({
    "Feature": df.columns,
    "Missing %": missing_percent,
    "Non-Missing %": non_missing_percent
}).sort_values(by="Missing %", ascending=False)  # Sort by highest missing

# Plot the histogram
fig, ax = plt.subplots(figsize=(10, 6))
bar_width = 0.5  # Adjust for better visibility

# Plot non-missing values below
ax.barh(missing_df["Feature"], missing_df["Non-Missing %"], color='green', label='Non-Missing %')

# Plot missing values on top
ax.barh(missing_df["Feature"], missing_df["Missing %"], color='red', left=missing_df["Non-Missing %"], label='Missing %')

# Add labels to bars
for i, (missing, total) in enumerate(zip(missing_df["Missing %"], missing_df["Non-Missing %"])):
```

```
    ax.text(total + (missing / 2), i, f"{missing:.1f}%", ha='center', va='center', color='black')

# Labels and legend
ax.set_xlabel("Percentage")
ax.set_ylabel("Features")
ax.set_title("Missing vs Non-Missing Data Percentage")
ax.legend()

plt.gca().invert_yaxis()  # Invert Y-axis for readability
plt.show()
```

In [14]:
```
# Drop missing values in specific columns as they are less than 5% of total data
df.dropna(subset=['Power', 'SINR', 'cellPosErrorLambda1'], inplace=True)
```

In [15]:
```
# Fill missing values in UTC using linear interpolation
df['UTC'] = df['UTC'].interpolate(method='linear')
```

In [16]:
```
# Fill missing values in Altitude using the mean
df['Altitude'] = df['Altitude'].fillna(df['Altitude'].mean())
```

In [17]:
```
# Fill missing values in Speed using forward fill
df['Speed'] = df['Speed'].ffill()
```

In [18]:
```
# Re-check for missing values
print("\nMissing Values:")
print(df.isnull().sum())
```

```
Missing Values:
Date                  0
Time                  0
UTC                   0
Latitude              0
Longitude             0
Altitude              0
Speed                 0
EARFCN                0
Frequency             0
PCI                   0
MNC                   0
CellIdentity          0
eNodeB.ID             0
Power                 0
SINR                  0
RSRP                  0
RSRQ                  0
scenario              0
cellLongitude         0
cellLatitude          0
cellPosErrorLambda1   0
cellPosErrorLambda2   0
n_CellIdentities      0
distance              0
Band                  0
campaign              0
dtype: int64
```

In [19]:
```
# Summary statistics for numerical columns
print("\nSummary Statistics:")
print(df.describe())
```

```
Summary Statistics:
                UTC       Latitude      Longitude       Altitude  \
count  4.989480e+05  498948.000000  498948.000000  498948.000000
mean   1.611763e+09      41.883311      12.485922      69.665625
std    1.186063e+06       0.014223       0.022434      38.486457
min    1.610528e+09      41.823736      12.418497    -139.740000
25%    1.610726e+09      41.871591      12.464796      46.460000
50%    1.610980e+09      41.890558      12.494067      69.665625
75%    1.612689e+09      41.893862      12.495064      85.190000
max    1.614353e+09      41.903102      12.533860     416.550000

               Speed         EARFCN      Frequency            PCI  \
count  498948.000000  498948.000000  498948.000000  498948.000000
mean        5.118639    2878.254546    1851.325515     223.932253
std         9.306316    2102.771914     656.199727     143.094869
min         0.000000     501.000000     806.000000       1.000000
25%         0.680000    1225.000000    1807.500000     103.000000
50%         2.560000    1850.000000    1870.000000     197.000000
75%         4.680000    3175.000000    2647.500000     322.000000
max        79.270000    6400.000000    2662.500000     503.000000

       CellIdentity      eNodeB.ID  ...           SINR           RSRP  \
count  4.989480e+05  498948.000000  ...  498948.000000  498948.000000
mean   4.457121e+07  174106.165789  ...       1.013055     -98.673668
std    2.922670e+07  114166.814599  ...      10.582073      14.131144
min    1.691931e+07   66091.000000  ...     -20.930000    -145.520000
25%    1.771959e+07   69217.000000  ...      -8.310000    -108.140000
50%    1.776797e+07   69406.000000  ...       2.880000     -98.720000
75%    7.684507e+07  300176.000000  ...       7.370000     -89.220000
max    7.745152e+07  302545.000000  ...      39.020000     -43.170000

                RSRQ  cellLongitude   cellLatitude  cellPosErrorLambda1  \
count  498948.000000  498948.000000  498948.000000        498948.000000
mean      -19.912534      12.486287      41.883034            12.587258
std         5.001063       0.022452       0.015768            13.481647
min       -49.890000      12.418621      41.824584             0.139538
25%       -23.650000      12.467131      41.871271             0.361032
50%       -19.150000      12.493153      41.890689            10.810001
75%       -15.690000      12.497913      41.896503            20.130005
max        -7.990000      12.543056      41.902997           100.000000

       cellPosErrorLambda2  n_CellIdentities       distance           Band
count        498948.000000     498948.000000  498948.000000  498948.000000
mean             12.516248          7.688304     553.247336       7.681251
std              13.539498          2.957037     785.299550       7.089620
min               0.116809          1.000000       0.000000       1.000000
25%               0.248061          5.000000     158.577997       3.000000
50%              10.810001          8.000000     288.167398       3.000000
75%              20.130005         10.000000     617.165332       7.000000
max             100.000000         13.000000    8788.724127      20.000000

[8 rows x 21 columns]
```

In [20]:
```
# Shows the total number of rows and columns within the dataset
df.shape
```

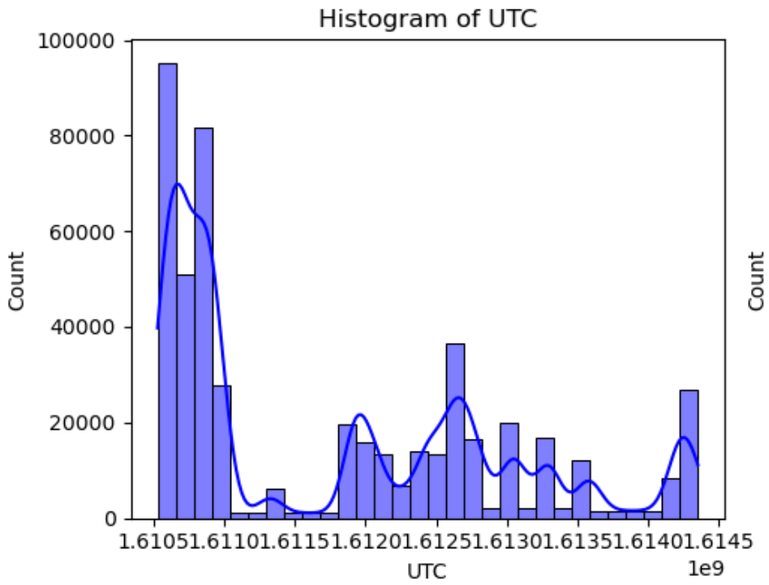
Out[20]: (498948, 26)

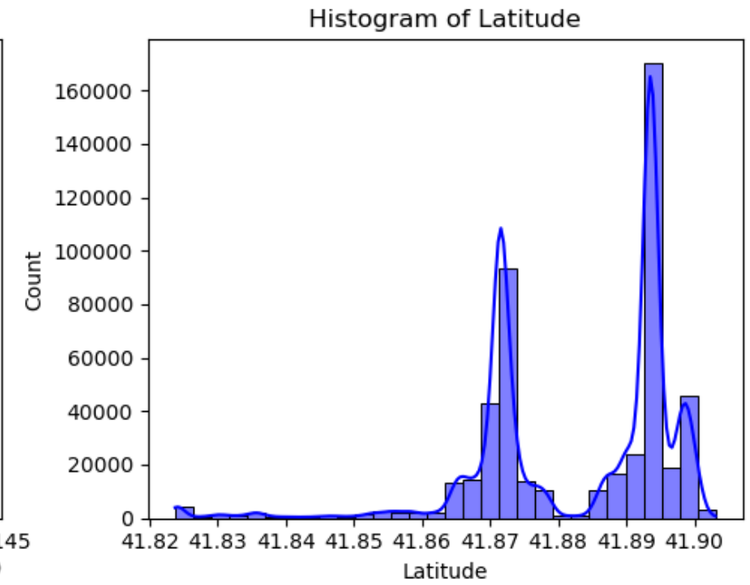
In [21]:
```python
def plot_normality_check(df, columns, bins=30):
    """
    Plots histograms for specified numerical columns to check for normality
    and calculates skewness and kurtosis.

    Parameters:
    df (DataFrame): The dataset containing the columns to plot.
    columns (list): List of column names to plot.
    bins (int): Number of bins for the histogram (default: 30).
    """
    num_cols = len(columns)
    cols_per_row = 3  # Number of histograms per row (adjustable)
    rows = (num_cols + cols_per_row - 1) // cols_per_row  # Calculate required rows

    plt.figure(figsize=(5 * cols_per_row, 4 * rows))  # Adjust figure size dynamically

    for idx, col in enumerate(columns, 1):
        plt.subplot(rows, cols_per_row, idx)
        sns.histplot(df[col], bins=bins, kde=True, color='blue')
        plt.title(f'Histogram of {col}')
        plt.xlabel(col)
        plt.ylabel('Count')

        # Calculate and print skewness and kurtosis
        skewness = df[col].skew()
        kurtosis = df[col].kurtosis()
        print(f"{col} - Skewness: {skewness:.2f}, Kurtosis: {kurtosis:.2f}")

        if abs(skewness) > 1:
            print(f"{col} has a highly skewed distribution.\n")
        elif abs(skewness) > 0.5:
            print(f"{col} has a moderately skewed distribution.\n")
        else:
            print(f"{col} is approximately normally distributed.\n")

    plt.tight_layout()
    plt.show()

# List of numeric columns to check distribution
numeric_columns = [
    'UTC', 'Latitude', 'Longitude', 'Altitude', 'Speed',
    'EARFCN', 'Frequency', 'PCI', 'CellIdentity', 'eNodeB.ID',
    'Power', 'SINR', 'RSRP', 'RSRQ',
    'cellLongitude', 'cellLatitude',
    'cellPosErrorLambda1', 'cellPosErrorLambda2',
    'n_CellIdentities', 'distance', 'Band'
]

# Call the function with a consistent number of bins
plot_normality_check(df, numeric_columns, bins=30)
```

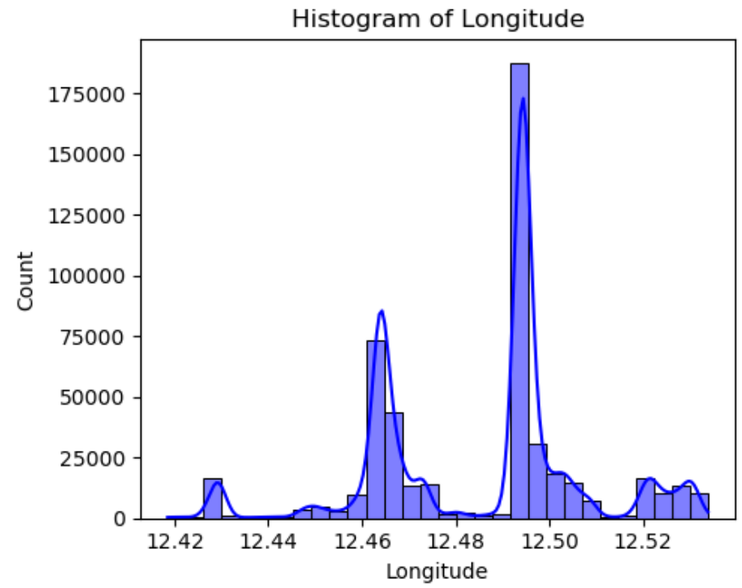
```
UTC - Skewness: 0.68, Kurtosis: -0.84
UTC has a moderately skewed distribution.
```

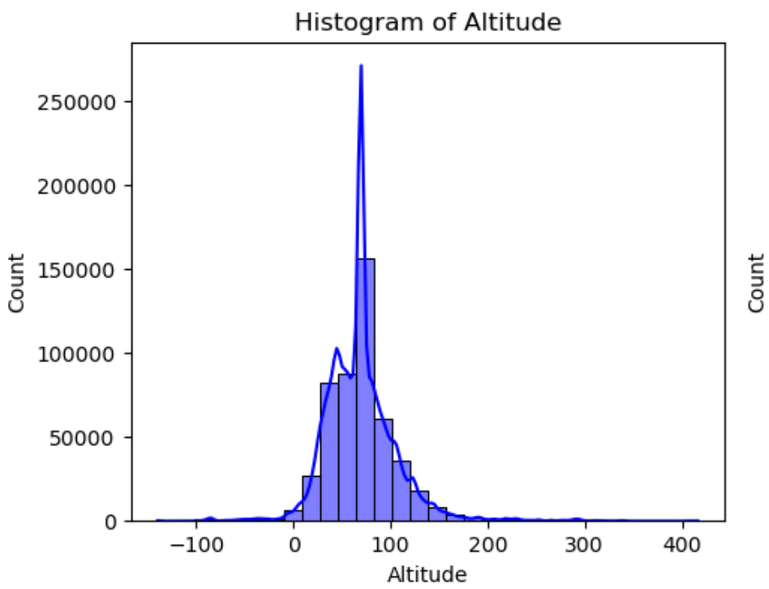
```
Latitude - Skewness: -1.22, Kurtosis: 2.12
Latitude has a highly skewed distribution.

Longitude - Skewness: -0.31, Kurtosis: 0.05
Longitude is approximately normally distributed.

Altitude - Skewness: 1.39, Kurtosis: 8.62
Altitude has a highly skewed distribution.

Speed - Skewness: 3.66, Kurtosis: 14.71
Speed has a highly skewed distribution.

EARFCN - Skewness: 0.72, Kurtosis: -0.93
EARFCN has a moderately skewed distribution.

Frequency - Skewness: -0.45, Kurtosis: -0.95
Frequency is approximately normally distributed.

PCI - Skewness: 0.29, Kurtosis: -1.02
PCI is approximately normally distributed.

CellIdentity - Skewness: 0.18, Kurtosis: -1.95
CellIdentity is approximately normally distributed.

eNodeB.ID - Skewness: 0.18, Kurtosis: -1.95
eNodeB.ID is approximately normally distributed.

Power - Skewness: -0.02, Kurtosis: -0.21
Power is approximately normally distributed.

SINR - Skewness: 0.05, Kurtosis: -0.34
SINR is approximately normally distributed.

RSRP - Skewness: 0.01, Kurtosis: -0.19
RSRP is approximately normally distributed.

RSRQ - Skewness: -0.48, Kurtosis: -0.61
RSRQ is approximately normally distributed.

cellLongitude - Skewness: -0.39, Kurtosis: 0.18
cellLongitude is approximately normally distributed.

cellLatitude - Skewness: -1.14, Kurtosis: 1.40
cellLatitude has a highly skewed distribution.

cellPosErrorLambda1 - Skewness: 1.47, Kurtosis: 3.80
cellPosErrorLambda1 has a highly skewed distribution.

cellPosErrorLambda2 - Skewness: 1.46, Kurtosis: 3.73
cellPosErrorLambda2 has a highly skewed distribution.

n_CellIdentities - Skewness: -0.17, Kurtosis: -0.93
n_CellIdentities is approximately normally distributed.

distance - Skewness: 3.72, Kurtosis: 18.23
distance has a highly skewed distribution.
```

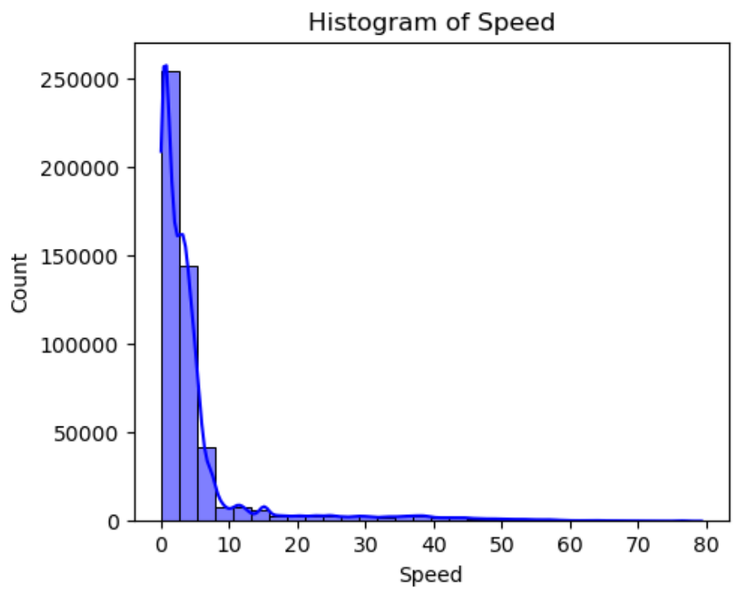
```
Band - Skewness: 0.99, Kurtosis: -0.68
Band has a moderately skewed distribution.
```

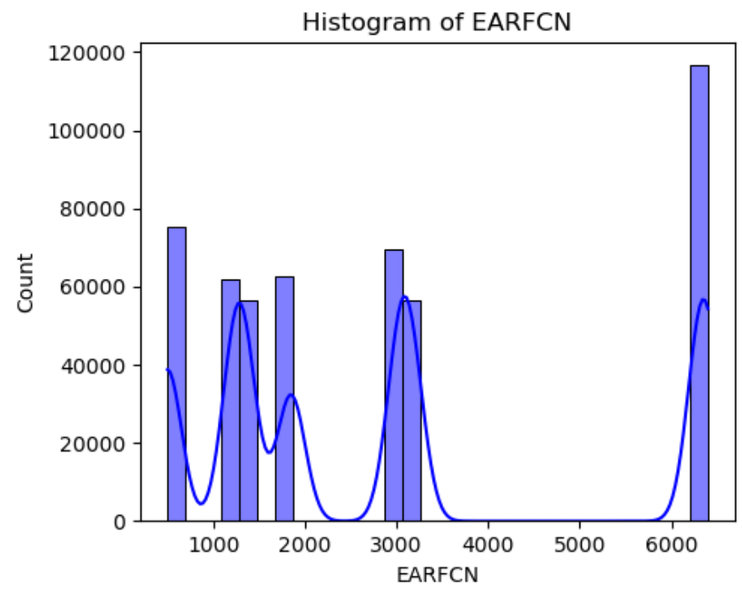


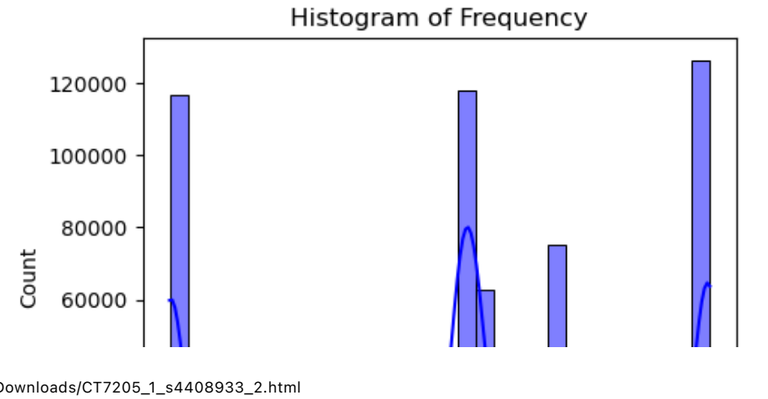


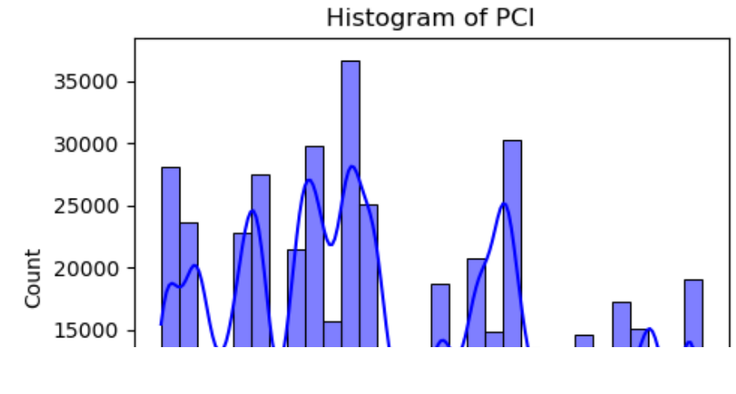


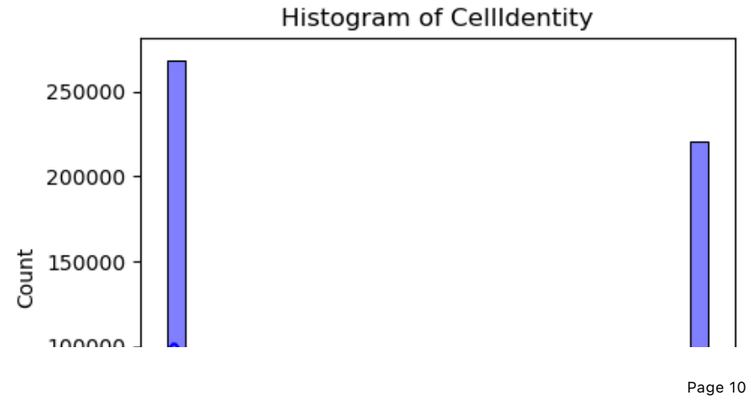


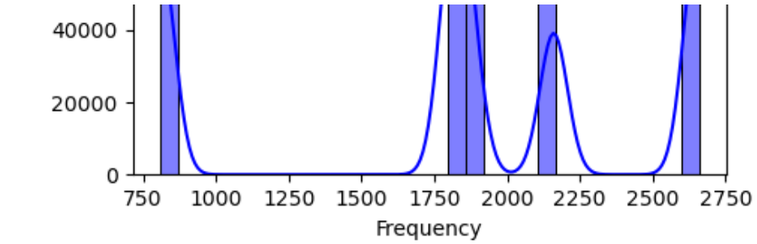


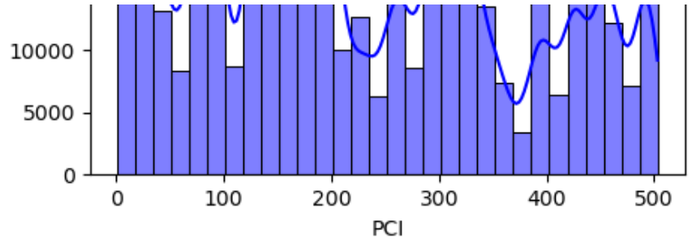


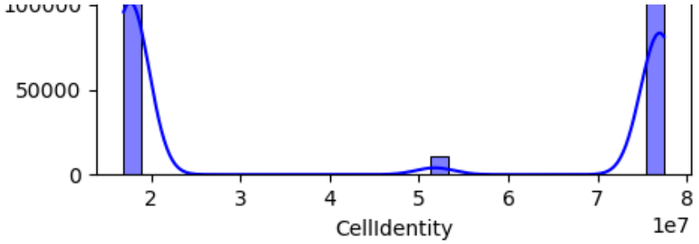


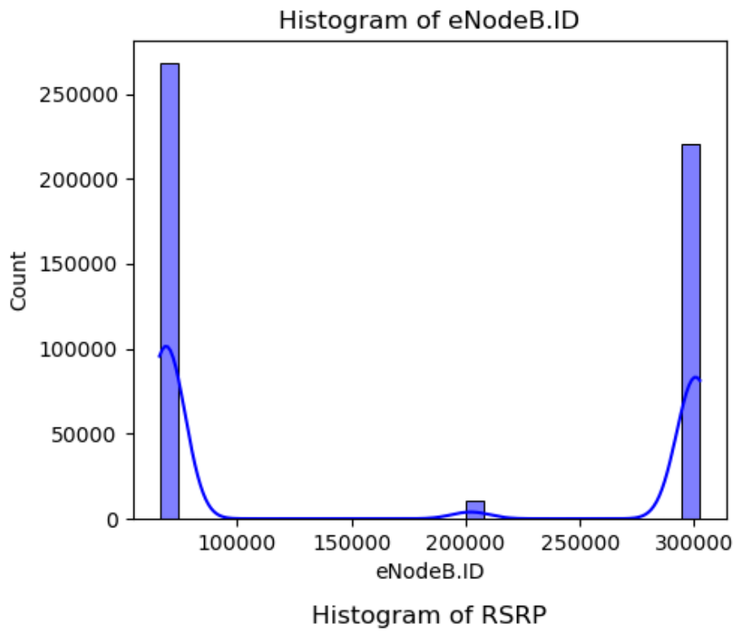


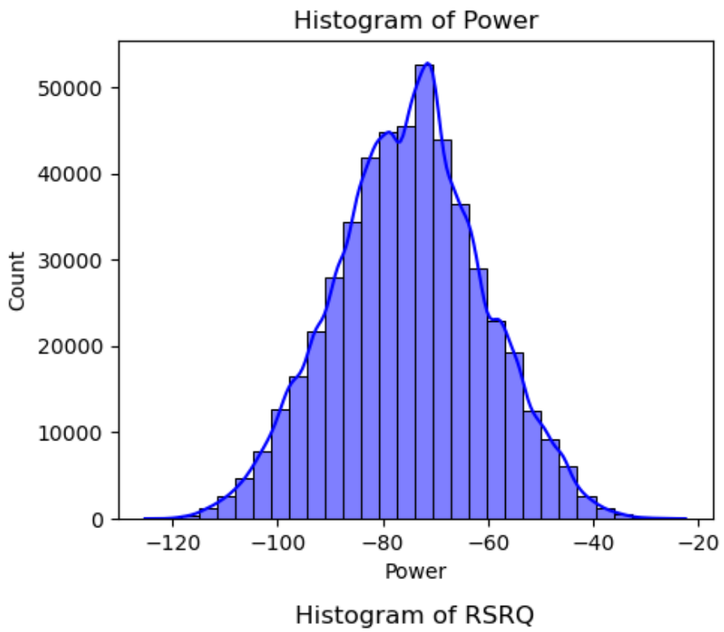


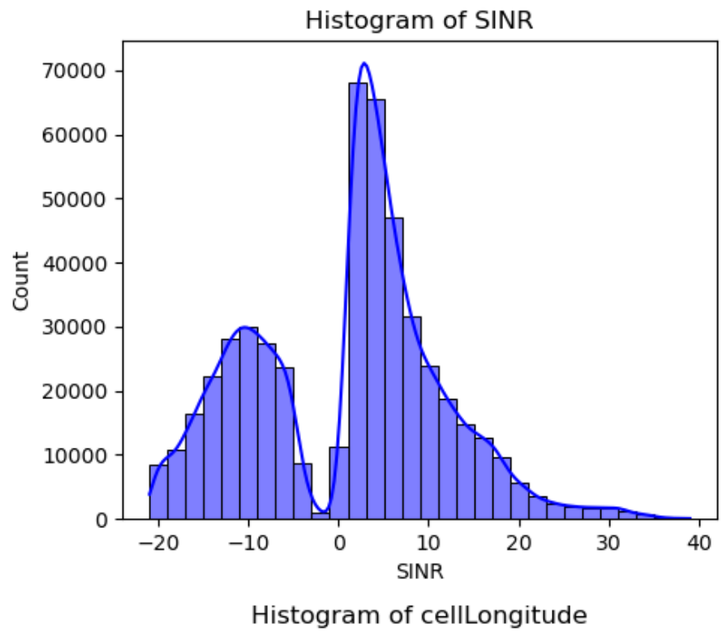

40000
20000
0
750 1000 1250 1500 1750 2000 2250 2500 2750
Frequency

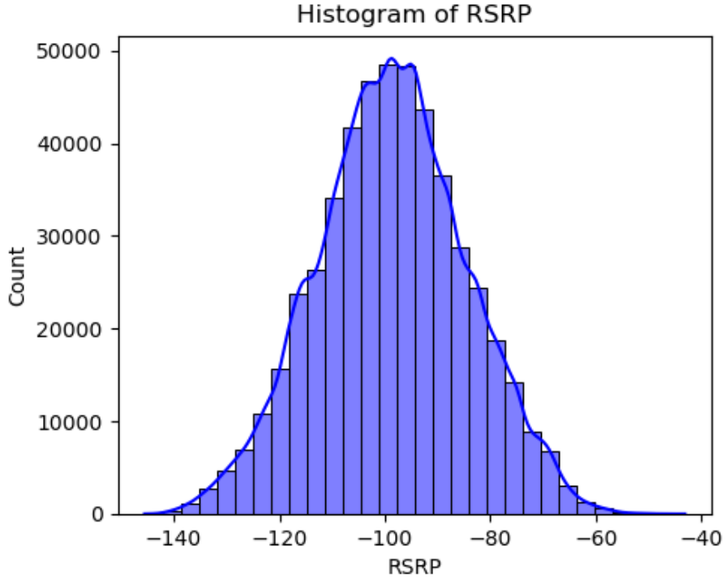

10000
5000
0
0 100 200 300 400 500
PCI

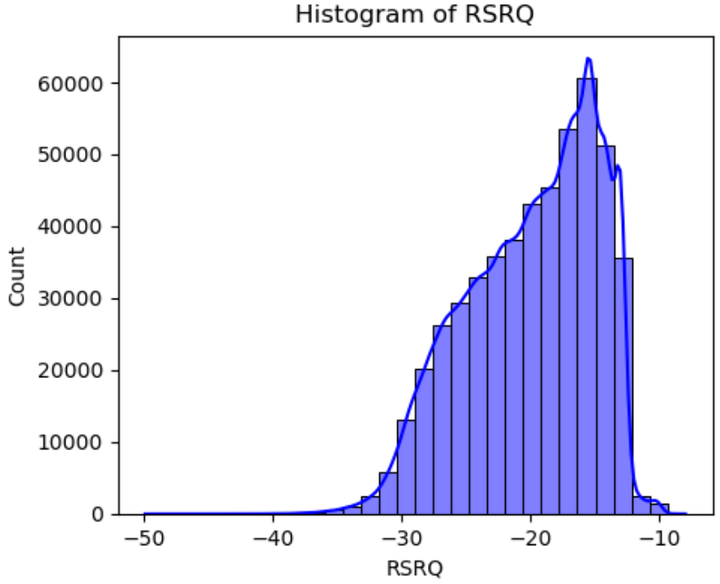

50000
0
2 3 4 5 6 7 8
CellIdentity
1e7

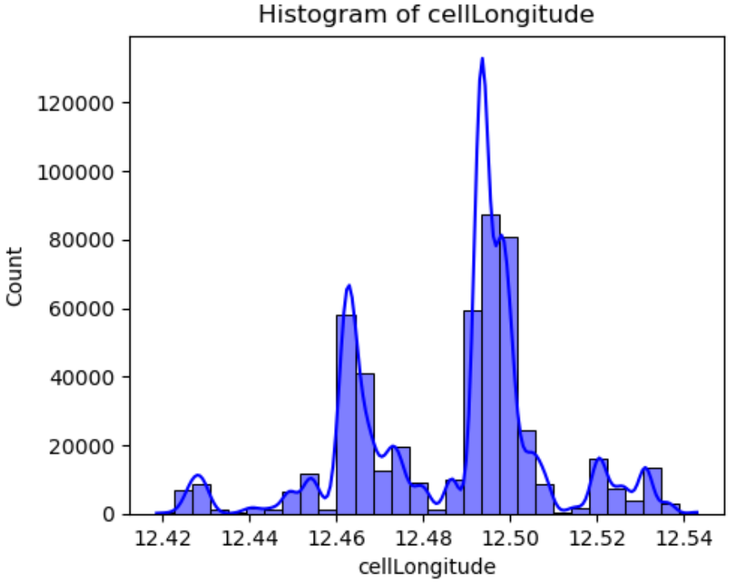

Histogram of eNodeB.ID
Count
250000
200000
150000
100000
50000
0
100000 150000 200000 250000 300000
eNodeB.ID


Histogram of Power
Count
50000
40000
30000
20000
10000
0
−120 −100 −80 −60 −40 −20
Power


Histogram of SINR
Count
70000
60000
50000
40000
30000
20000
10000
0
−20 −10 0 10 20 30 40
SINR


Histogram of RSRP
Count
50000
40000
30000
20000
10000
0
−140 −120 −100 −80 −60 −40
RSRP

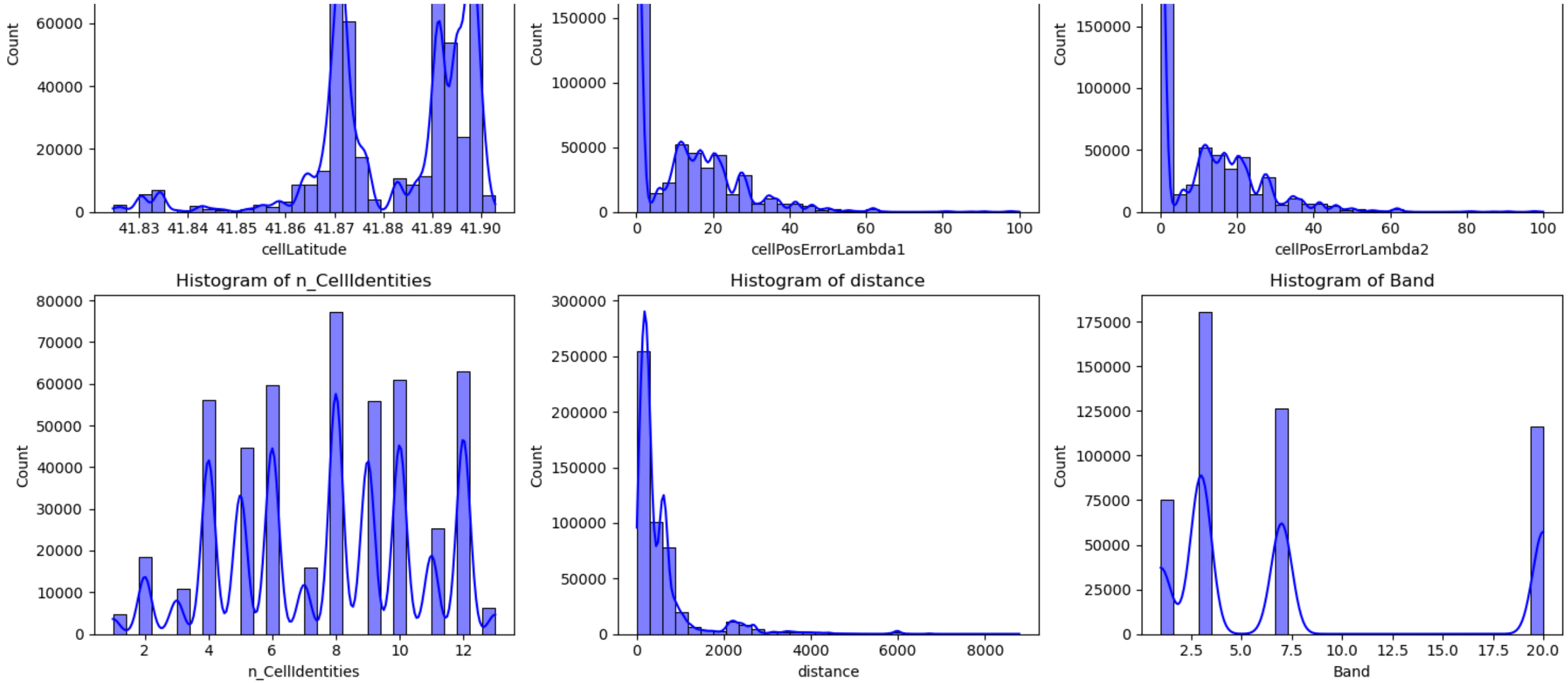

Histogram of RSRQ
Count
60000
50000
40000
30000
20000
10000
0
−50 −40 −30 −20 −10
RSRQ


Histogram of cellLongitude
Count
120000
100000
80000
60000
40000
20000
0
12.42 12.44 12.46 12.48 12.50 12.52 12.54
cellLongitude

Histogram of cellLatitude
100000
80000

Histogram of cellPosErrorLambda1
250000
200000

Histogram of cellPosErrorLambda2
250000
200000



In [22]:
```python
def detect_outliers_iqr(df, column, threshold=1.5):
    """Detect outliers using the IQR method."""
    Q1 = df[column].quantile(0.25)
    Q3 = df[column].quantile(0.75)
    IQR = Q3 - Q1
    lower_bound = Q1 - (threshold * IQR)
    upper_bound = Q3 + (threshold * IQR)
    outliers = df[(df[column] < lower_bound) | (df[column] > upper_bound)]
    return outliers

def detect_outliers_zscore(df, column, threshold=3):
    """Detect outliers using the Z-score method."""
    mean = df[column].mean()
    std = df[column].std()
    z_scores = (df[column] - mean) / std
    outliers = df[np.abs(z_scores) > threshold]
    return outliers

# Columns grouped by distribution type from your analysis
moderately_skewed_columns = ['UTC', 'EARFCN', 'Band', 'cellPosErrorLambda1', 'cellPosErrorLambda2']
highly_skewed_columns = ['Latitude', 'cellLatitude', 'Speed', 'distance']
normally_distributed_columns = ['Longitude', 'Altitude', 'Frequency', 'PCI',
```

```
                              'Power', 'SINR', 'RSRP', 'RSRQ',
                              'cellLongitude', 'n_CellIdentities', 'CellIdentity', 'eNodeB.ID']

# Detect outliers
outlier_results = {}

# IQR for Moderately Skewed (Threshold 1.5 IQR)
for col in moderately_skewed_columns:
    outliers = detect_outliers_iqr(df, col, threshold=1.5)
    outlier_results[col] = len(outliers)

# IQR for Highly Skewed (Threshold 3.0 IQR)
for col in highly_skewed_columns:
    outliers = detect_outliers_iqr(df, col, threshold=3.0)
    outlier_results[col] = len(outliers)

# Z-Score for Normally Distributed Columns
for col in normally_distributed_columns:
    outliers = detect_outliers_zscore(df, col, threshold=3)
    outlier_results[col] = len(outliers)

# Display Outlier Results
outlier_results_df = pd.DataFrame(list(outlier_results.items()), columns=['Column', 'Outlier_Count'])

# Display the DataFrame using the print function
print(outlier_results_df.to_string(index=False))
```

```
           Column  Outlier_Count
              UTC              0
           EARFCN         116583
             Band         116583
cellPosErrorLambda1           7942
cellPosErrorLambda2           7942
         Latitude              0
      cellLatitude              0
            Speed          35388
         distance          34980
        Longitude            122
         Altitude           8914
        Frequency              0
              PCI              0
            Power            330
             SINR           1014
             RSRP            305
             RSRQ            721
    cellLongitude             75
  n_CellIdentities              0
      CellIdentity              0
         eNodeB.ID              0
```

In [23]:
```
# Given total dataset count
total_count = 498948

# Outlier counts per column
outlier_counts = {
    "UTC": 0,
```

```
    "EARFCN": 116583,
    "Band": 116583,
    "cellPosErrorLambda1": 7942,
    "cellPosErrorLambda2": 7942,
    "Latitude": 0,
    "cellLatitude": 0,
    "Speed": 35388,
    "distance": 34980,
    "Longitude": 122,
    "Altitude": 8914,
    "Frequency": 0,
    "PCI": 0,
    "Power": 330,
    "SINR": 1014,
    "RSRP": 305,
    "RSRQ": 721,
    "cellLongitude": 75,
    "n_CellIdentities": 0,
    "CellIdentity": 0,
    "eNodeB.ID": 0
}

# Calculate percentage of outliers for each column
outlier_percentages = {col: (count / total_count) * 100 for col, count in outlier_counts.items()}

# Create DataFrame for structured reference
df_outliers = pd.DataFrame(list(outlier_percentages.items()), columns=["Column", "Outlier_Percentage"])

# Display updated outlier percentages
print(df_outliers)
```

```
                Column  Outlier_Percentage
0                  UTC            0.000000
1               EARFCN           23.365762
2                 Band           23.365762
3   cellPosErrorLambda1            1.591749
4   cellPosErrorLambda2            1.591749
5             Latitude            0.000000
6         cellLatitude            0.000000
7                Speed            7.092523
8             distance            7.010751
9            Longitude            0.024451
10            Altitude            1.786559
11           Frequency            0.000000
12                 PCI            0.000000
13               Power            0.066139
14                SINR            0.203228
15                RSRP            0.061129
16                RSRQ            0.144504
17       cellLongitude            0.015032
18    n_CellIdentities            0.000000
19        CellIdentity            0.000000
20           eNodeB.ID            0.000000
```

In [24]:
```
print("Value counts for 'Band' column:")
print(df["Band"].value_counts())
```

```
Value counts for 'Band' column:
Band
3     180819
7     126147
20    116583
1      75399
Name: count, dtype: int64
```

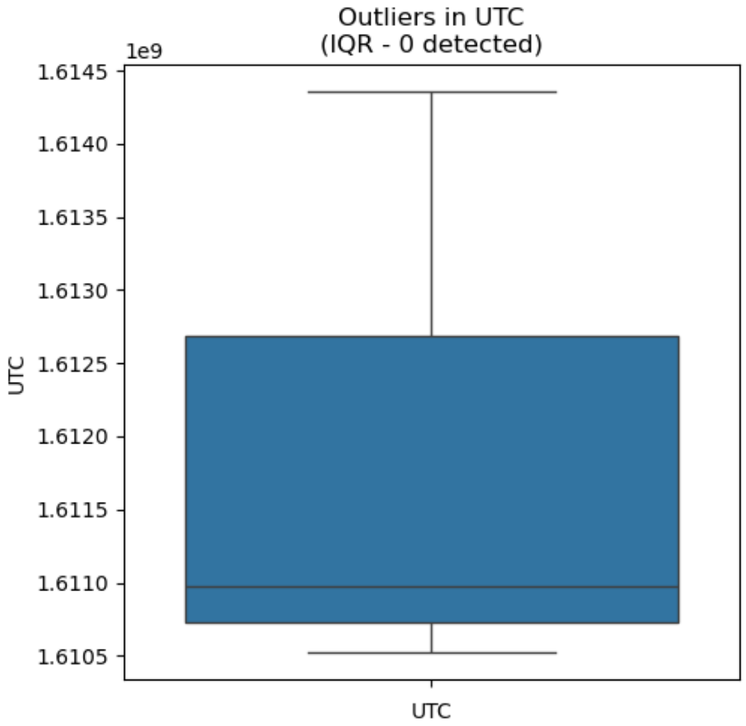
In [25]:
```
print("Value counts for 'EARFCN' column:")
print(df["EARFCN"].value_counts())
```

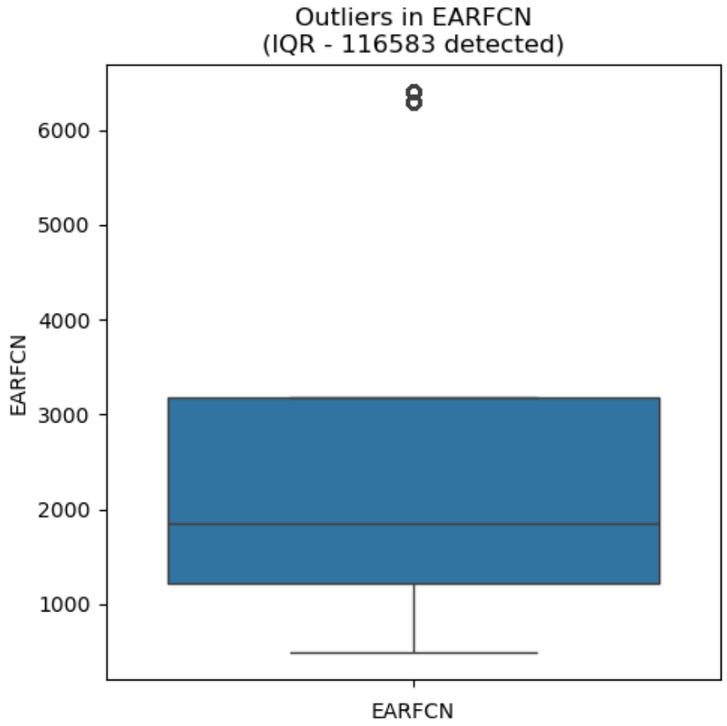
```
Value counts for 'EARFCN' column:
EARFCN
501     75399
3025    69653
1850    62720
1225    61739
6400    60345
3175    56494
1350    56360
6300    56238
Name: count, dtype: int64
```

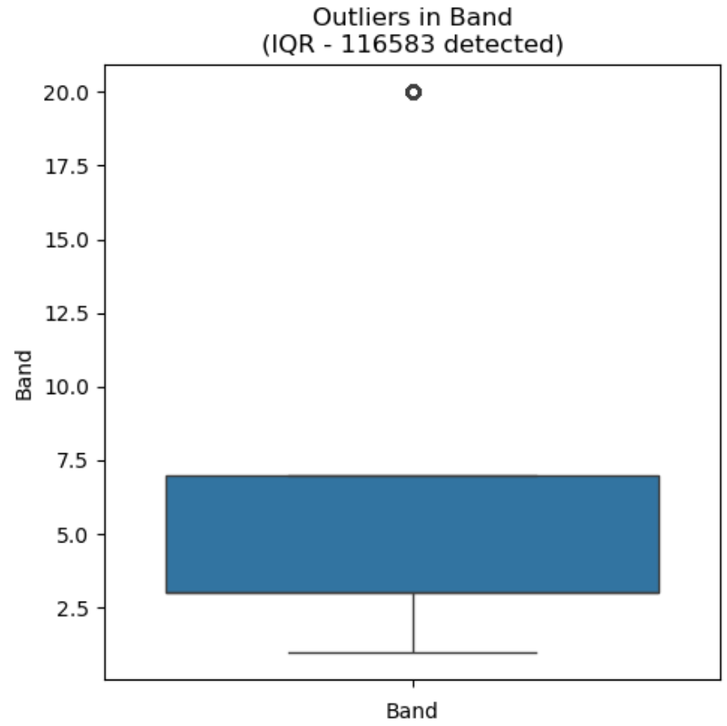
In [26]:
```
def visualize_outliers(df, columns, method='iqr', threshold=1.5, cols_per_row=3):
    """
    Visualize outliers for multiple columns using boxplots displayed side by side.

    Parameters:
        df (pd.DataFrame): The DataFrame containing the data.
    columns (list): List of column names to analyze.
    method (str): The method to detect outliers ('iqr' or 'zscore').
    threshold (float): The threshold for outlier detection.
    cols_per_row (int): Number of boxplots per row.
    """
    num_cols = len(columns)
    rows = (num_cols + cols_per_row - 1) // cols_per_row  # Calculate required rows

    plt.figure(figsize=(5 * cols_per_row, 5 * rows))  # Adjust figure size dynamically

    for idx, column in enumerate(columns, 1):
        plt.subplot(rows, cols_per_row, idx)

        # Detect outliers
        if method == 'iqr':
            outliers = detect_outliers_iqr(df, column, threshold)
        elif method == 'zscore':
            outliers = detect_outliers_zscore(df, column, threshold)
        else:
            raise ValueError("Method must be 'iqr' or 'zscore'.")

        # Boxplot
        sns.boxplot(data=df[column])
        plt.title(f"Outliers in {column}\n({method.upper()} - {len(outliers)} detected)")
        plt.xlabel(column)

        # Print outlier count
        print(f"Column: {column} | Outliers detected: {len(outliers)}")
```

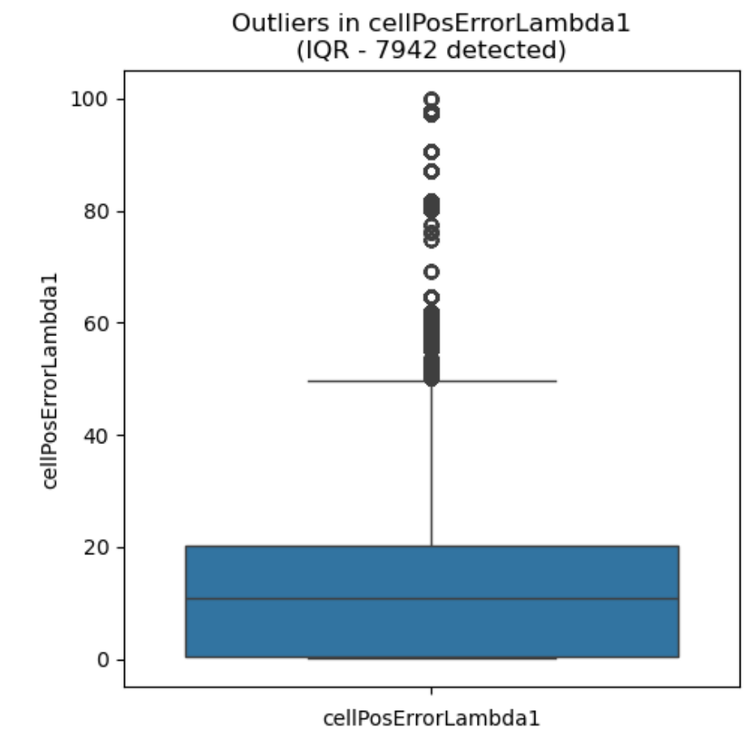
```
    plt.tight_layout()
    plt.show()

# Usage for different column groups
visualize_outliers(df, moderately_skewed_columns, method='iqr', threshold=1.5)  # Moderately skewed
visualize_outliers(df, highly_skewed_columns, method='iqr', threshold=3.0)  # Highly skewed
visualize_outliers(df, normally_distributed_columns, method='zscore', threshold=3)  # Normally distributed
```

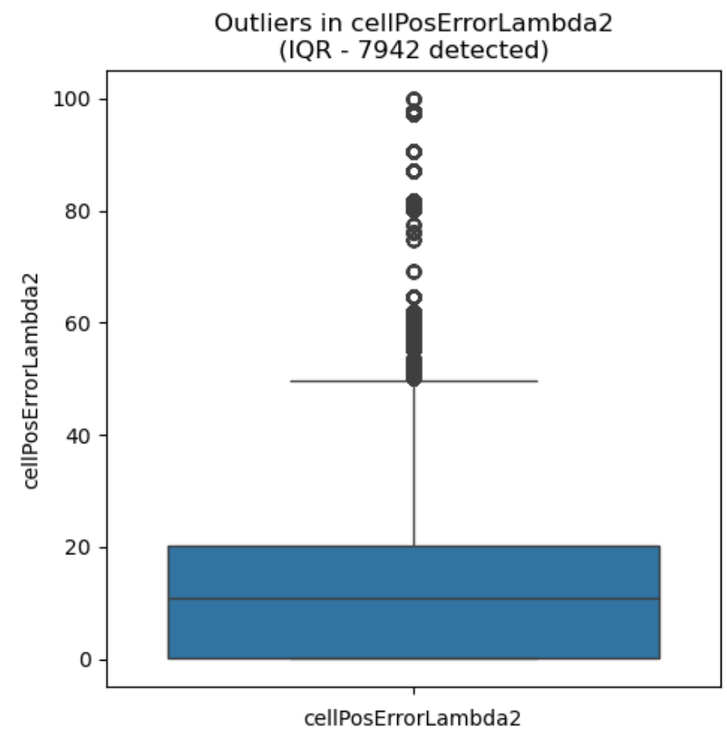
```
Column: UTC | Outliers detected: 0
Column: EARFCN | Outliers detected: 116583
Column: Band | Outliers detected: 116583
Column: cellPosErrorLambda1 | Outliers detected: 7942
Column: cellPosErrorLambda2 | Outliers detected: 7942
```



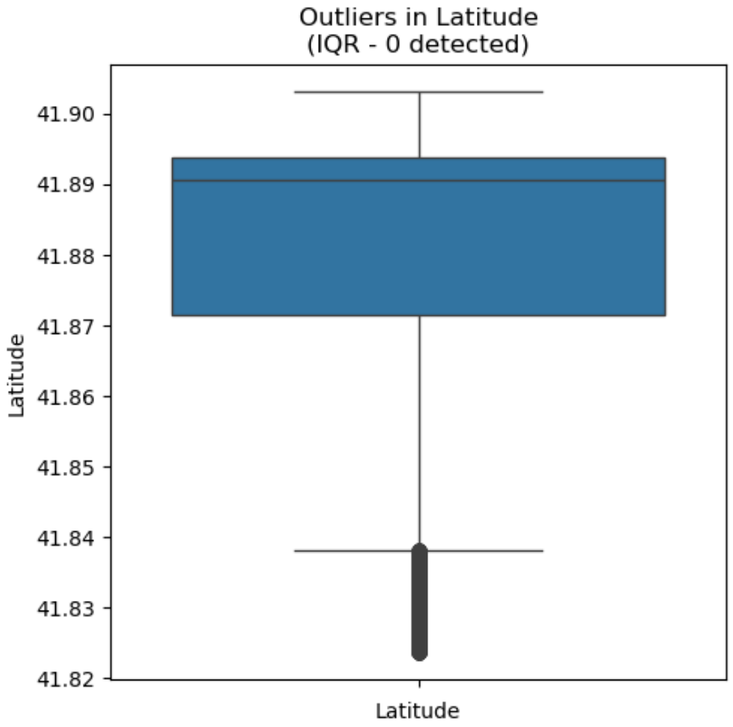


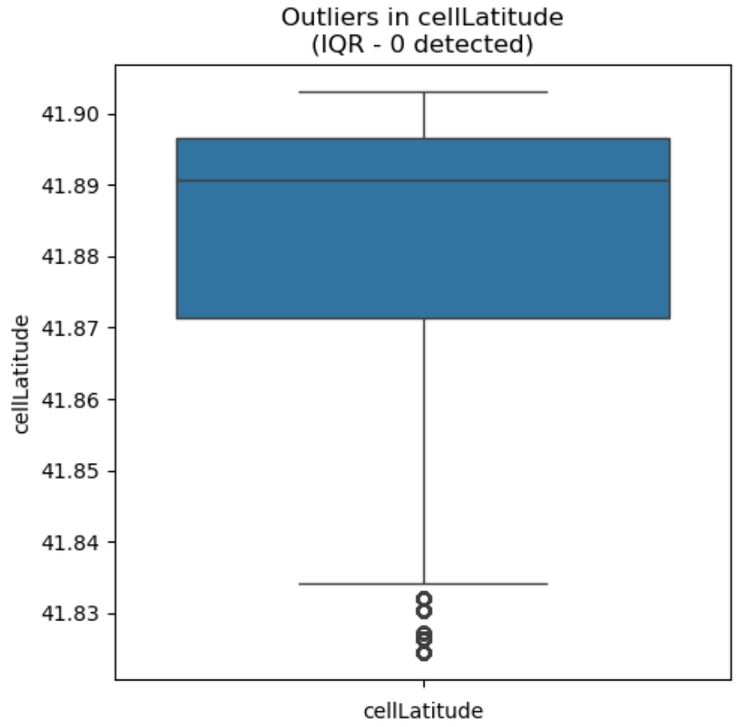


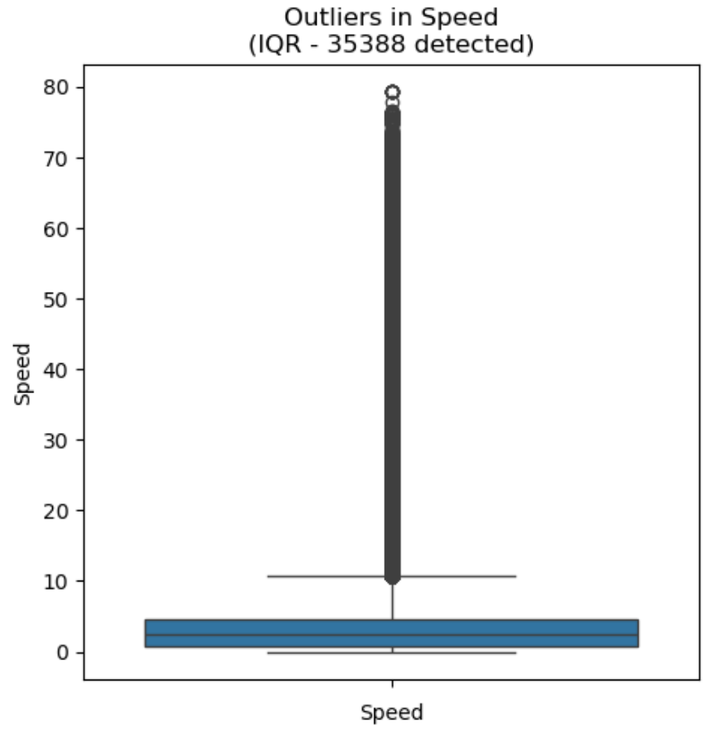


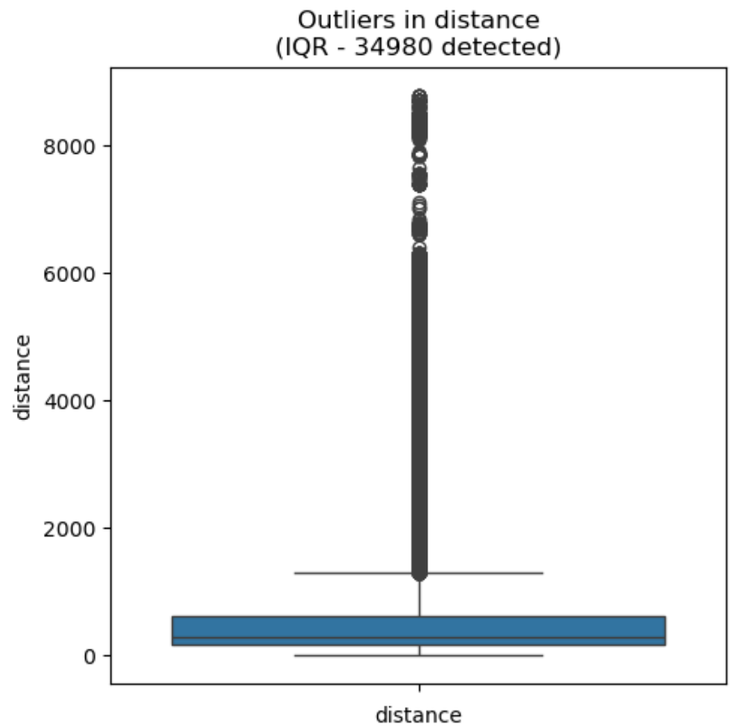



```
Column: Latitude | Outliers detected: 0
Column: cellLatitude | Outliers detected: 0
Column: Speed | Outliers detected: 35388
Column: distance | Outliers detected: 34980
```

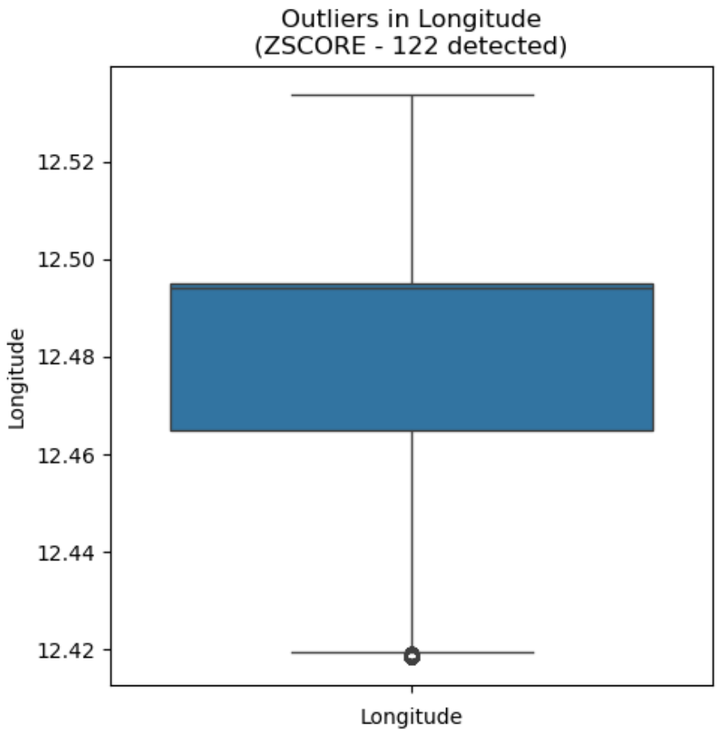

Outliers in Latitude
(IQR - 0 detected)
Latitude
41.90
41.89
41.88
41.87
41.86
41.85
41.84
41.83
41.82
Latitude

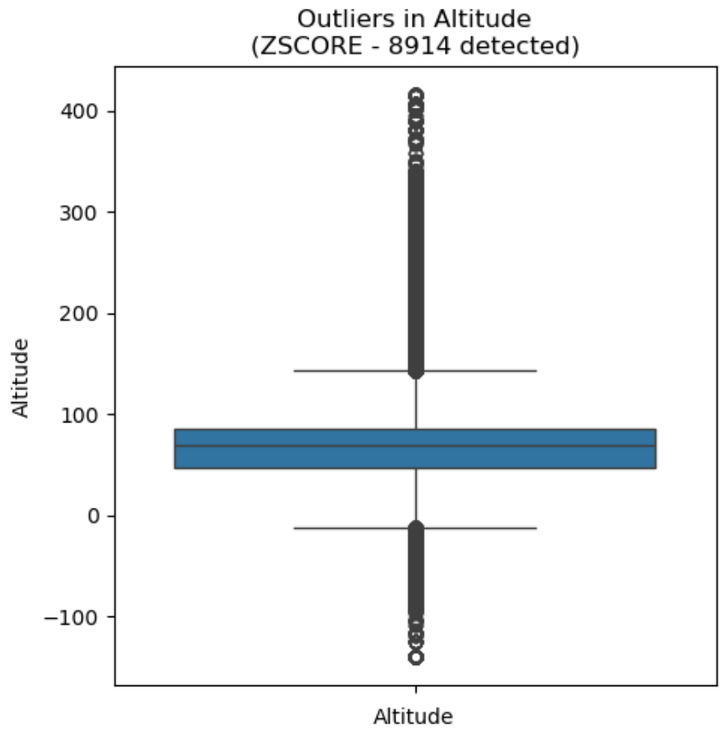

Outliers in cellLatitude
(IQR - 0 detected)
cellLatitude
41.90
41.89
41.88
41.87
41.86
41.85
41.84
41.83
cellLatitude

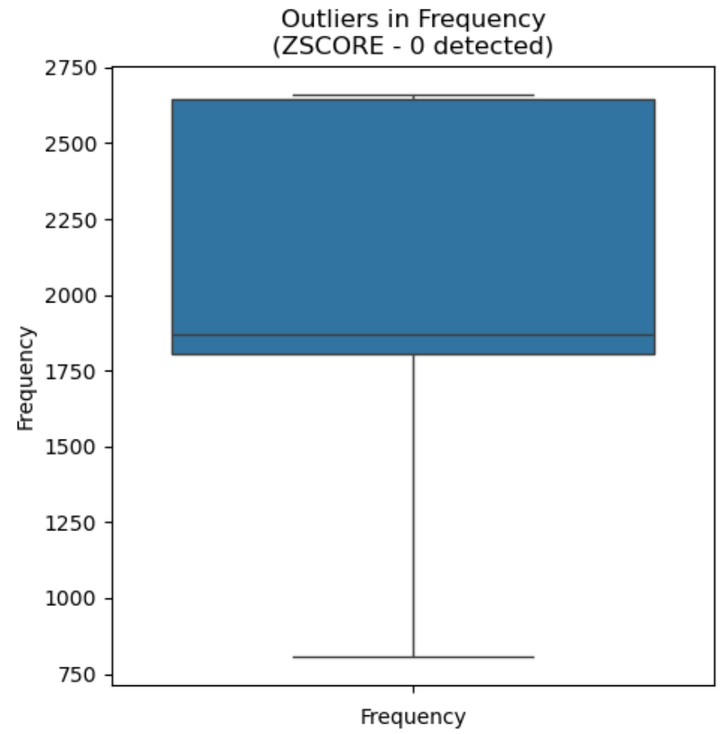

Outliers in Speed
(IQR - 35388 detected)
Speed
80
70
60
50
40
30
20
10
0
Speed

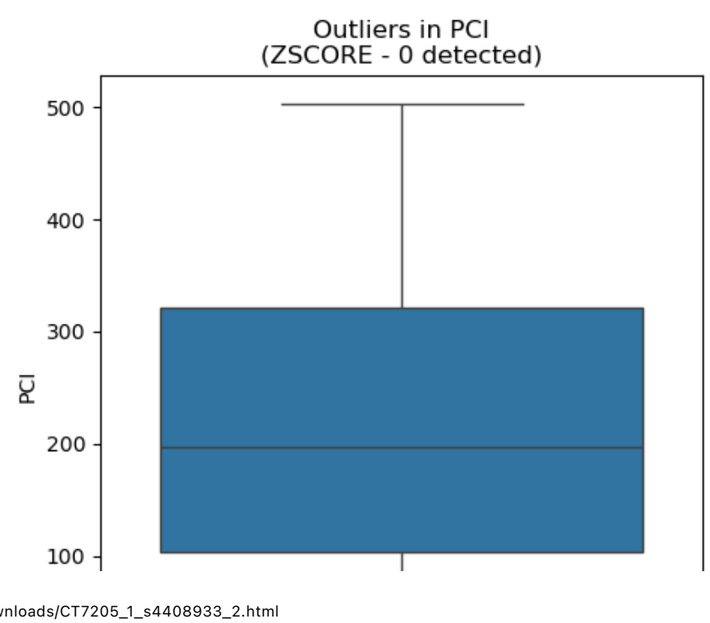

Outliers in distance
(IQR - 34980 detected)
distance
8000
6000
4000
2000
0
distance

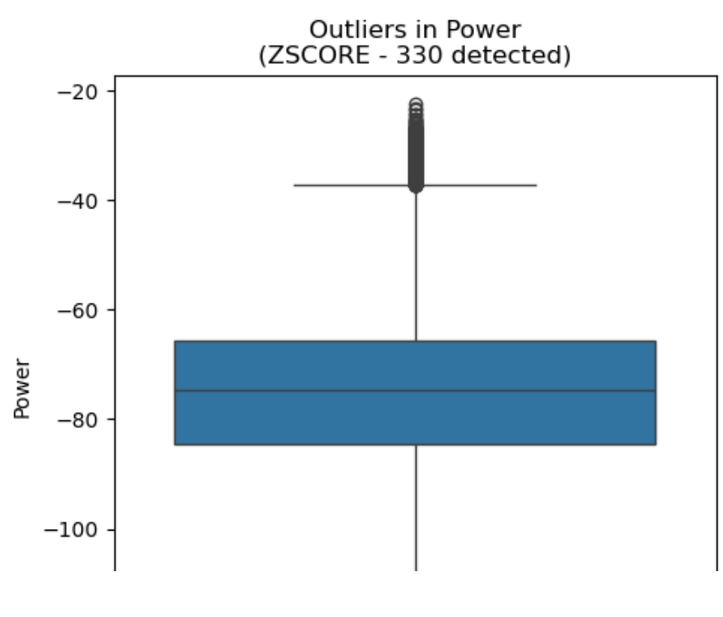
```
Column: Longitude | Outliers detected: 122
Column: Altitude | Outliers detected: 8914
Column: Frequency | Outliers detected: 0
Column: PCI | Outliers detected: 0
Column: Power | Outliers detected: 330
Column: SINR | Outliers detected: 1014
Column: RSRP | Outliers detected: 305
Column: RSRQ | Outliers detected: 721
Column: cellLongitude | Outliers detected: 75
Column: n_CellIdentities | Outliers detected: 0
Column: CellIdentity | Outliers detected: 0
Column: eNodeB.ID | Outliers detected: 0
```

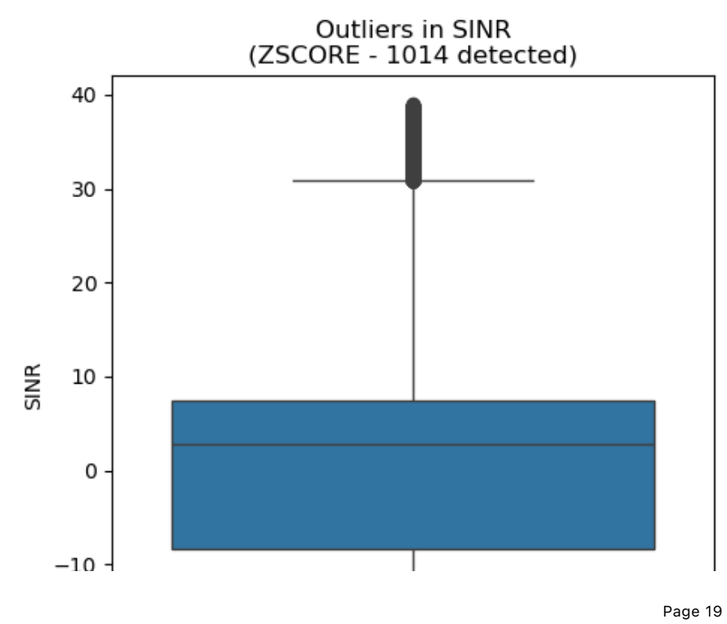


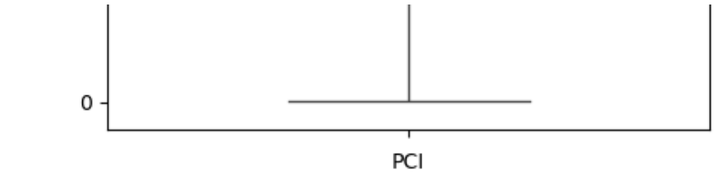


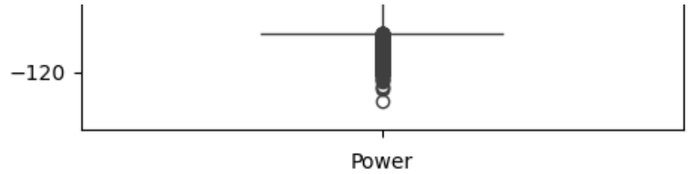


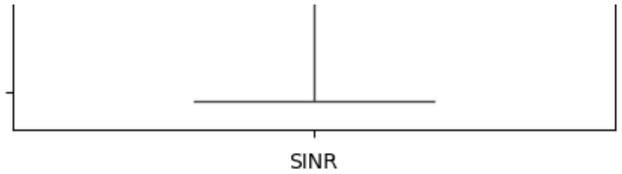


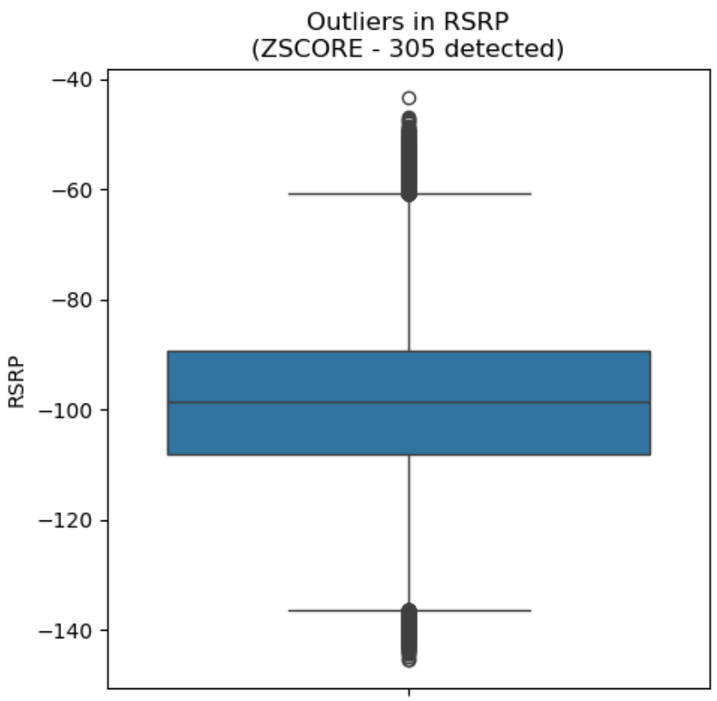


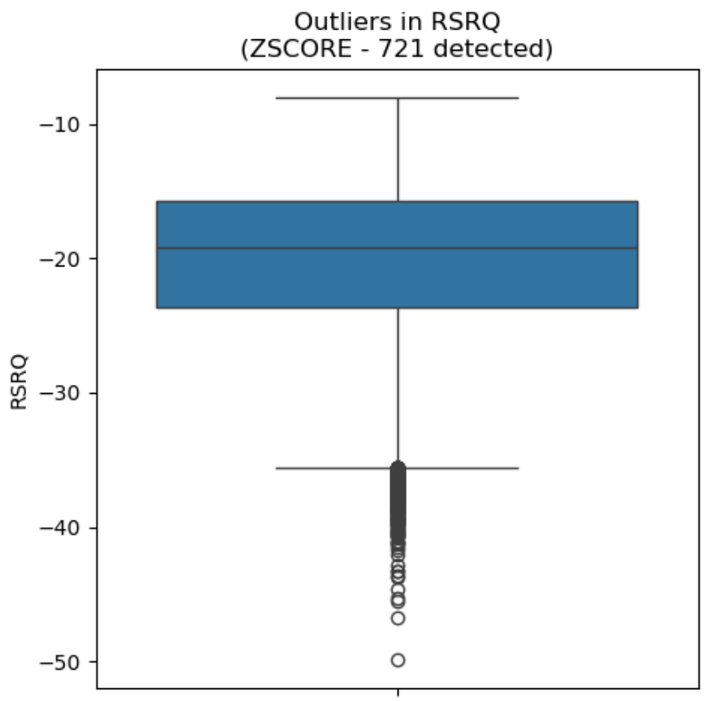


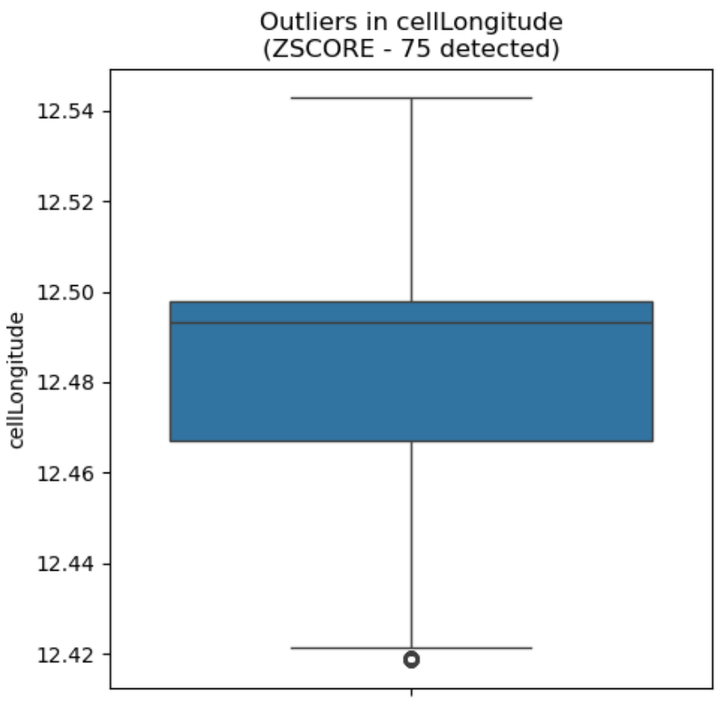




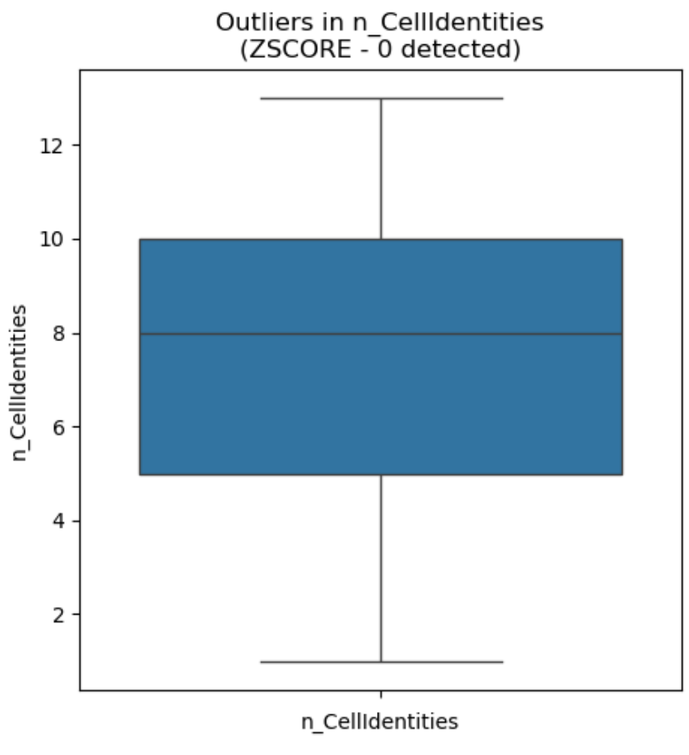


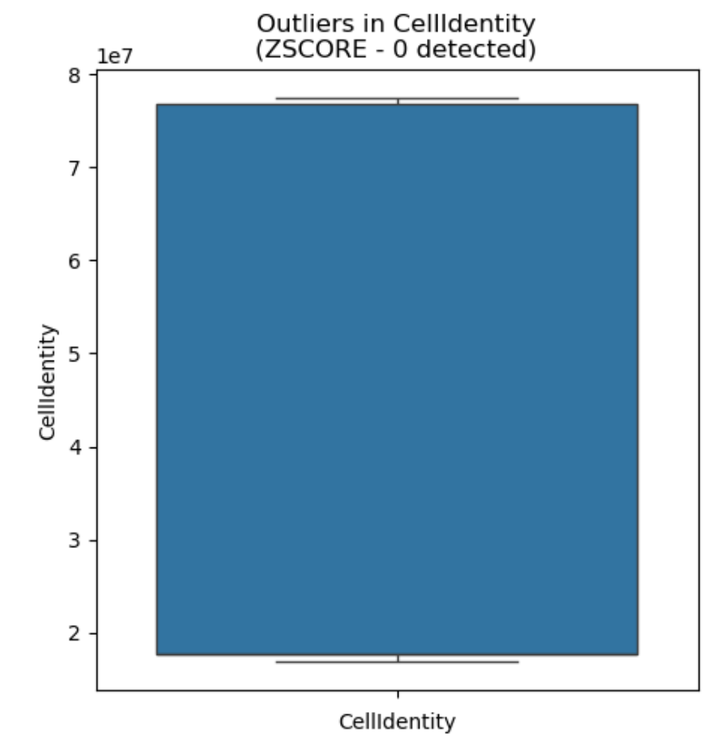


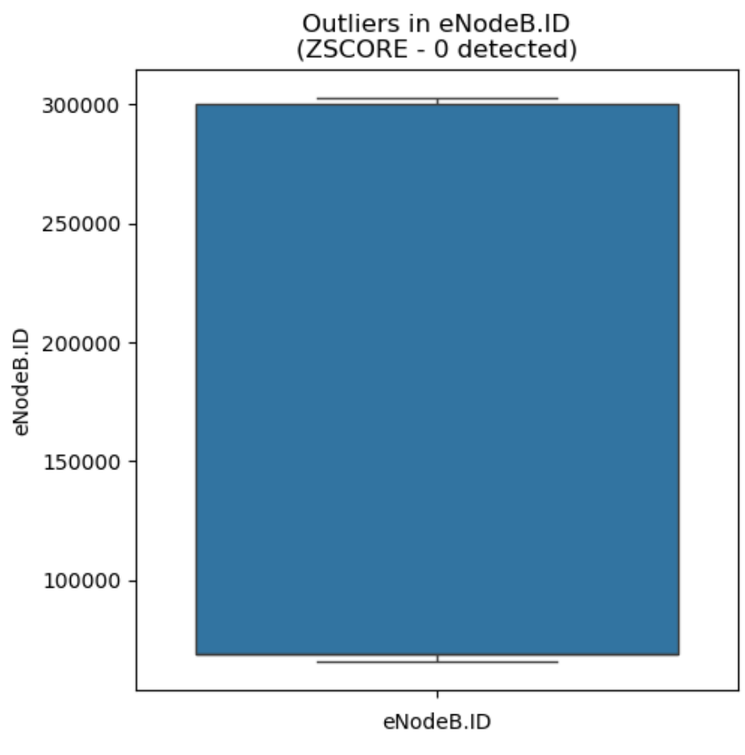




RSRP RSRQ cellLongitude







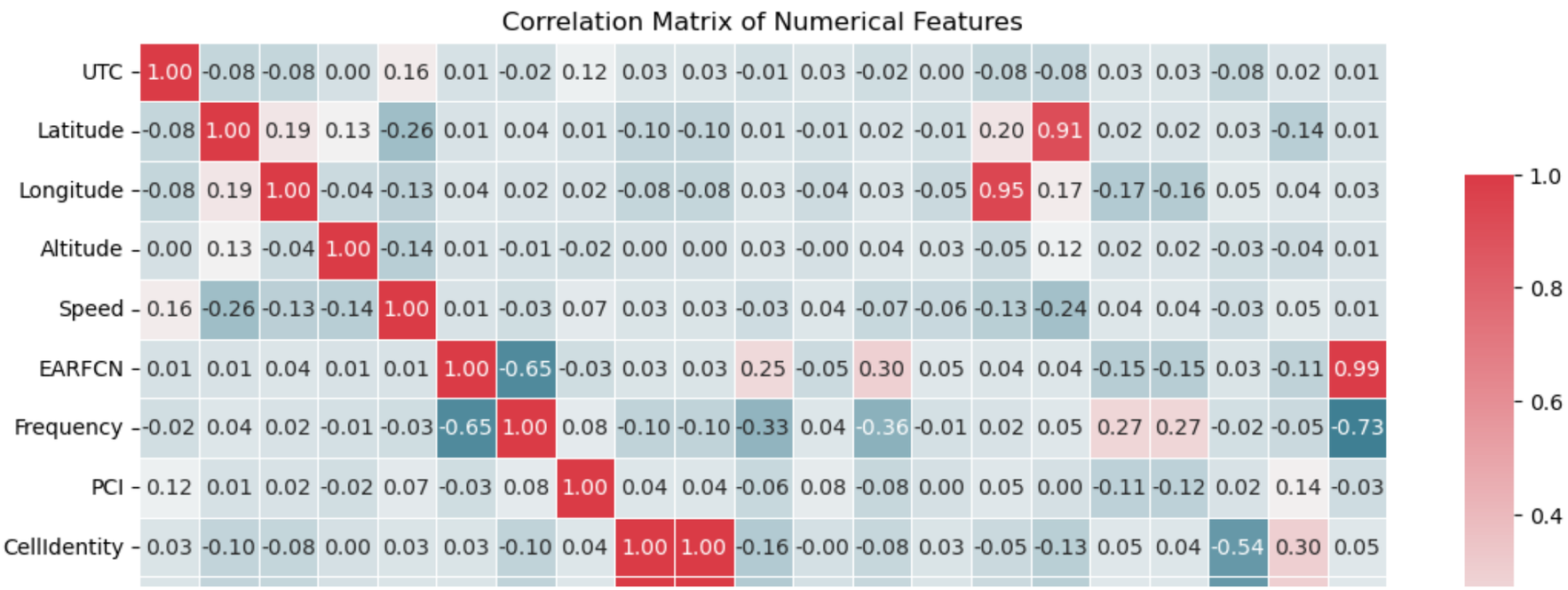
In [27]:
```
# Shows the total number of rows and columns within the dataset
df.shape
```

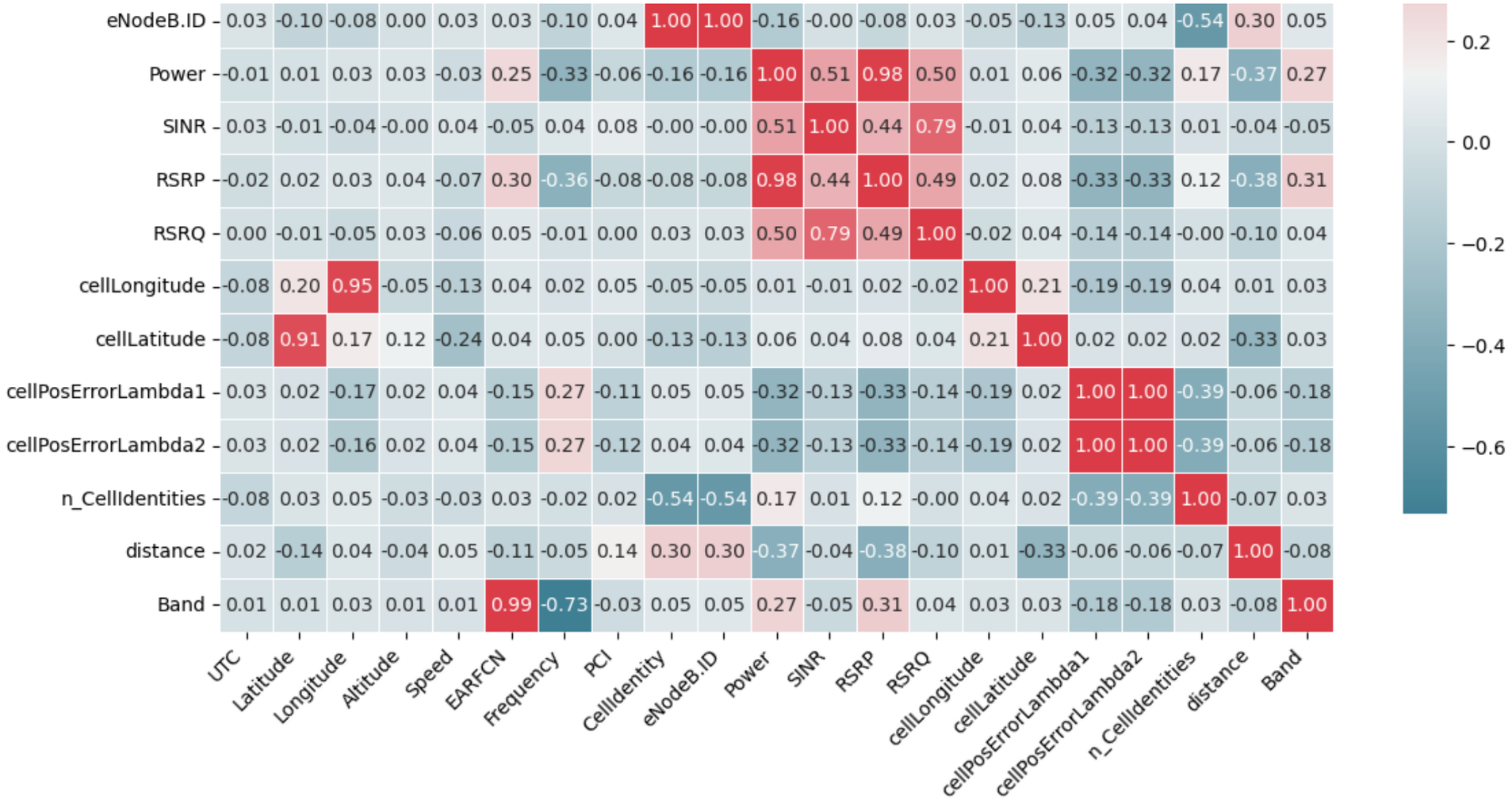
Out[27]: (498948, 26)

In [28]:
```
# Create a new DataFrame and save it as a cleaned dataset
df_cleaned = df.copy()
```

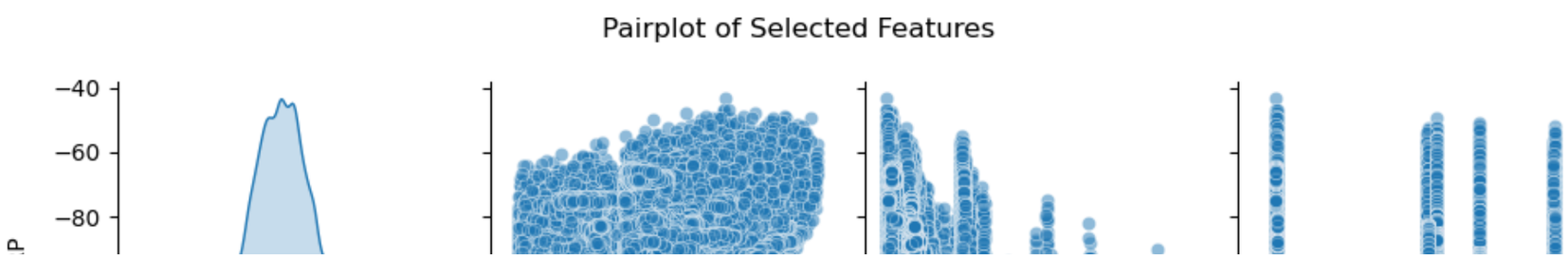
Exploratory Data Analysis

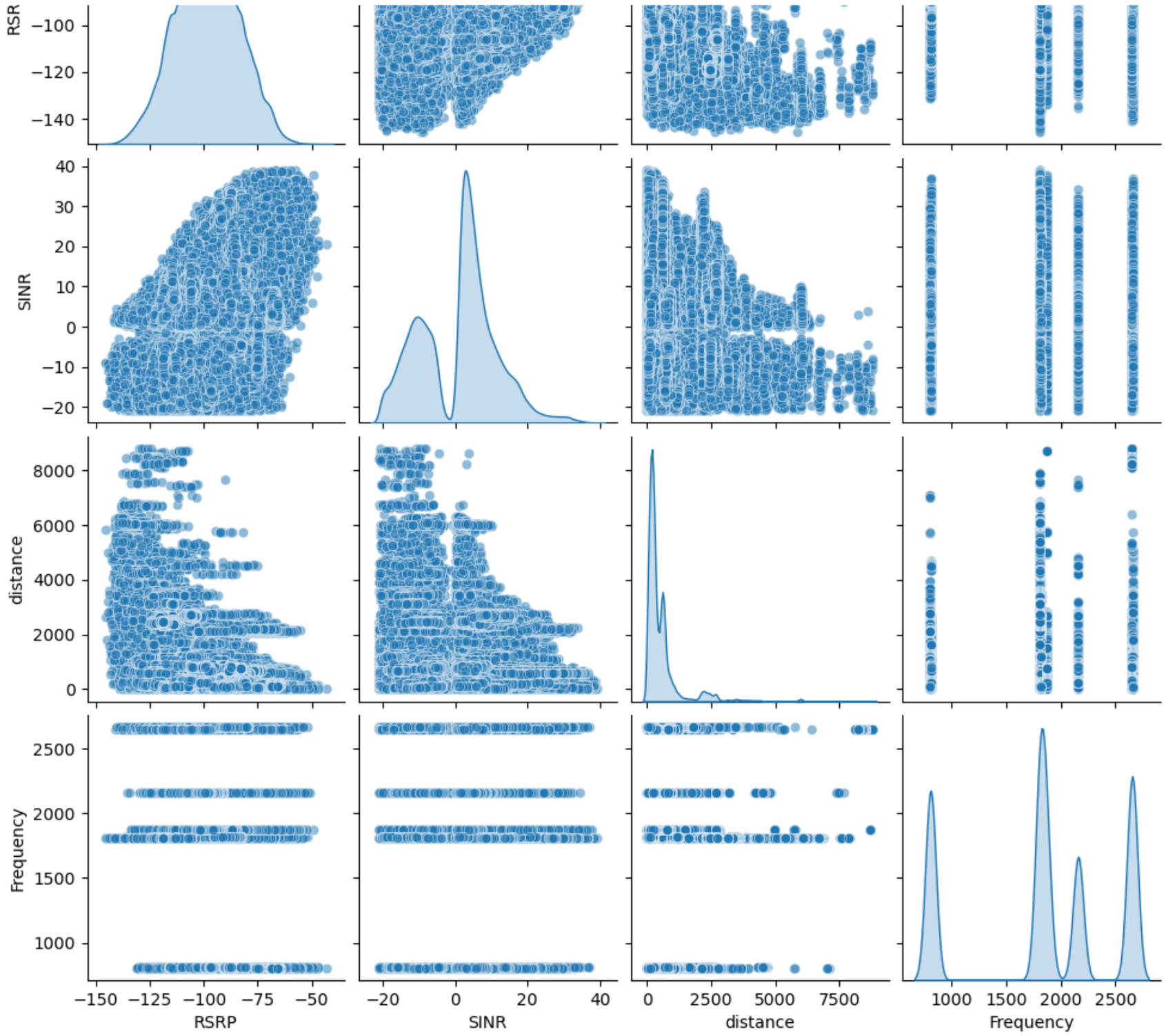
In [30]:
```
# Calculate correlations between numerical variables
numerical_features = df.select_dtypes(include=['number'])
corrmat = numerical_features.corr()
# Set up the matplotlib figure
fig, ax = plt.subplots(figsize=(12, 10))  # Adjust the size as needed
# Generate a custom diverging colormap
cmap = sns.diverging_palette(220, 10, as_cmap=True)
# Draw the heatmap with the full correlation matrix
sns.heatmap(corrmat, cmap=cmap, annot=True, fmt='.2f', linewidths=0.5,
            cbar_kws={"shrink": 0.75}, annot_kws={"size": 10},  # Adjust annotation font size
            square=True, ax=ax)
# Rotate the tick labels for better readability
plt.xticks(rotation=45, ha='right', fontsize=10)
plt.yticks(rotation=0, fontsize=10)
# Set the title with appropriate font size
plt.title('Correlation Matrix of Numerical Features', fontsize=12)
# Adjust layout to ensure everything fits without overlap
plt.tight_layout()
# Display the heatmap
plt.show()
```





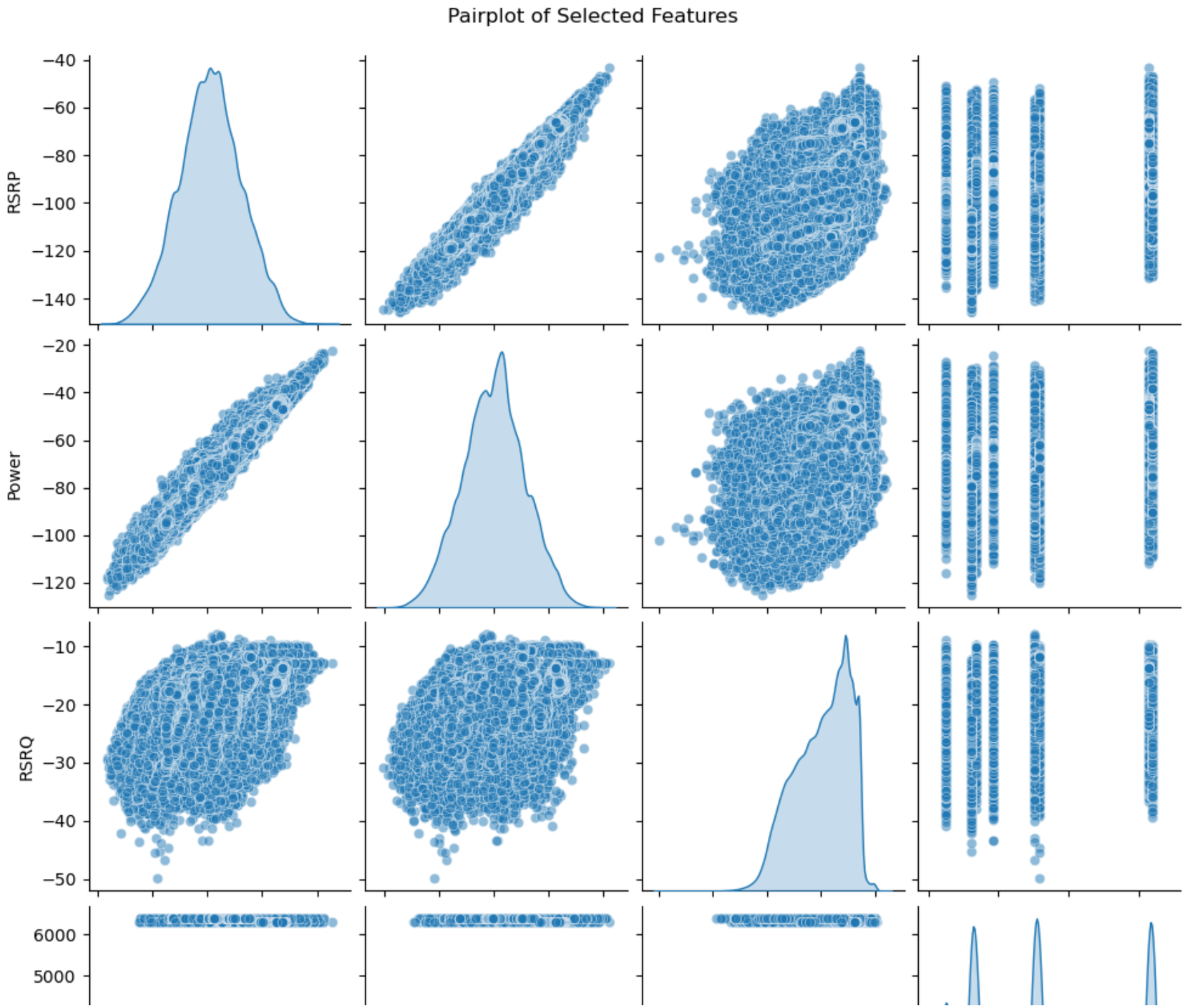
In [31]:
```python
# Define the features to include in the pairplot
features = ['RSRP', 'SINR', 'distance', 'Frequency']

# Create the pairplot
sns.pairplot(df[features], diag_kind='kde', plot_kws={'alpha': 0.5})

# Set the title for the entire plot
plt.suptitle('Pairplot of Selected Features', y=1.02)  # y=1.02 adjusts the title position

# Display the plot
plt.show()
```

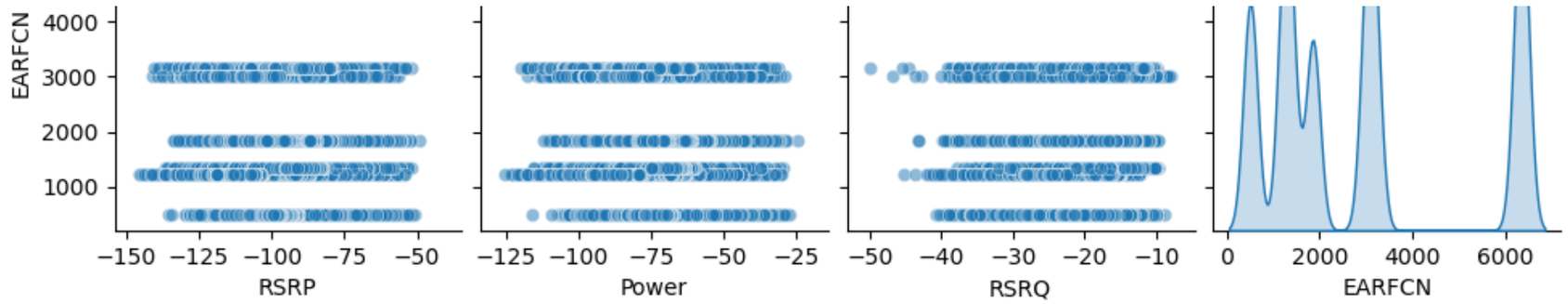




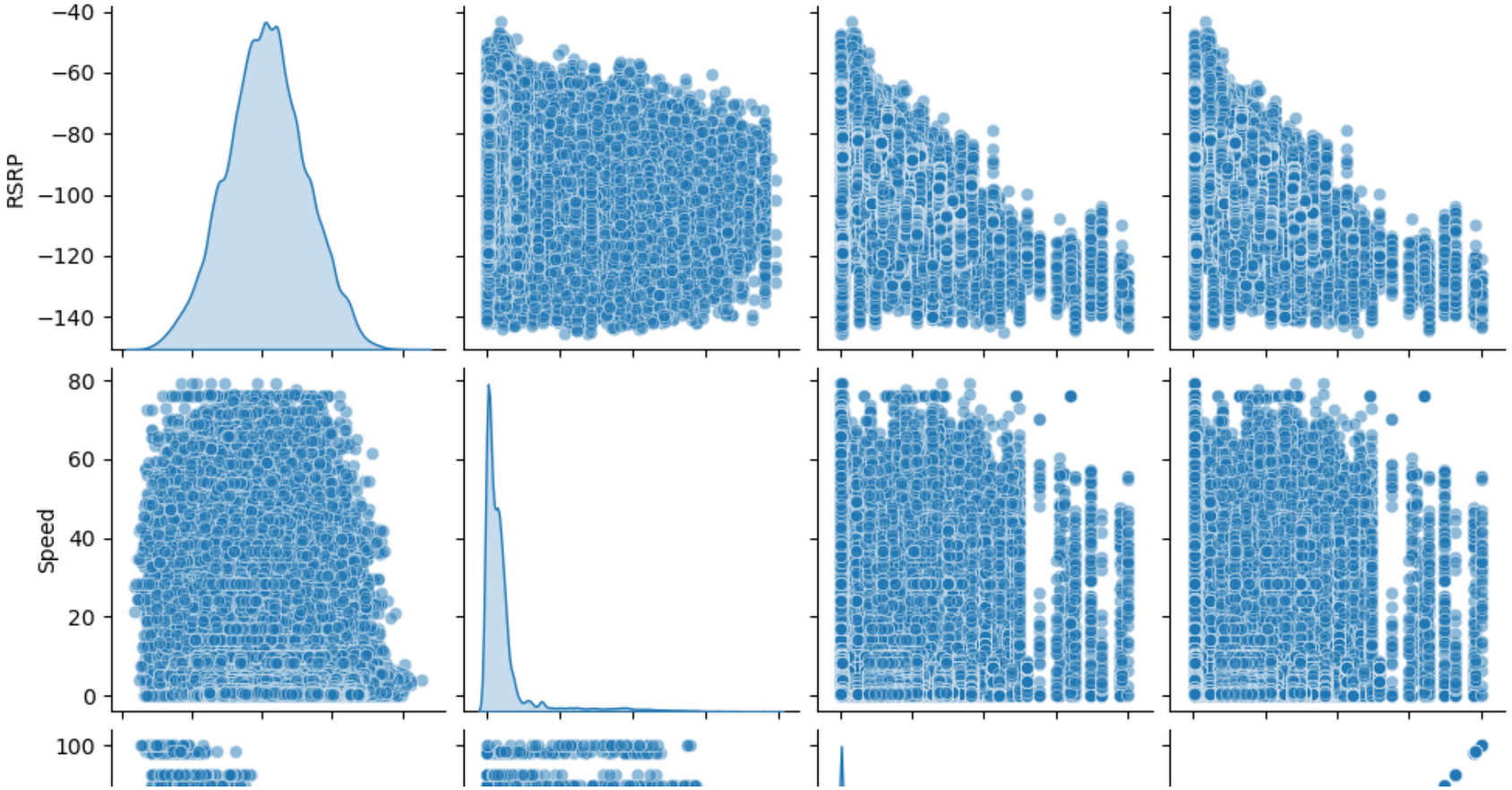
In [32]:
```
# Define the features to include in the pairplot
features = ['RSRP', 'Power', 'RSRQ', 'EARFCN']

# Create the pairplot
```

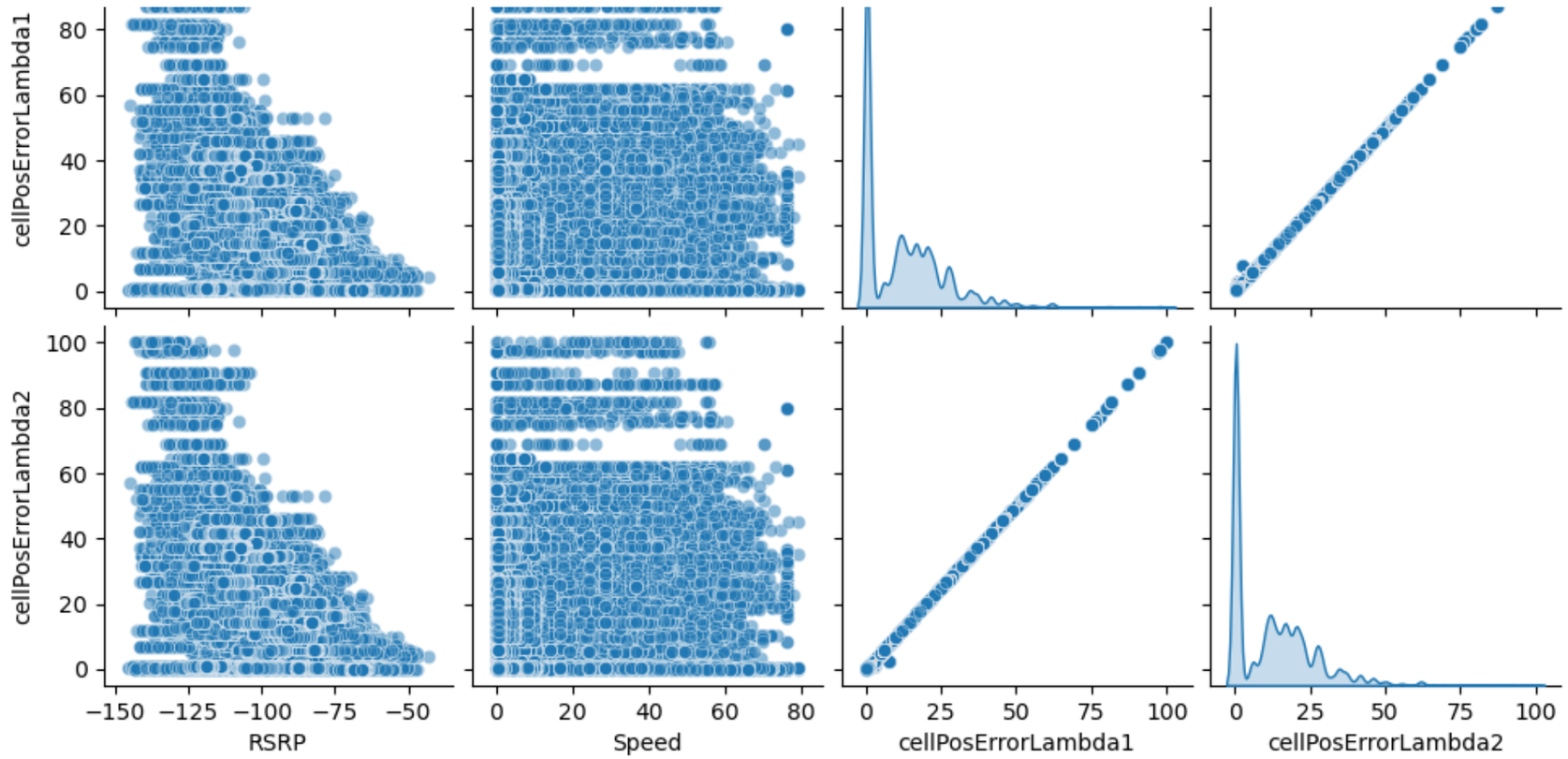
```
sns.pairplot(df[features], diag_kind='kde', plot_kws={'alpha': 0.5})

# Set the title for the entire plot
plt.suptitle('Pairplot of Selected Features', y=1.02)  # y=1.02 adjusts the title position

# Display the plot
plt.show()
```



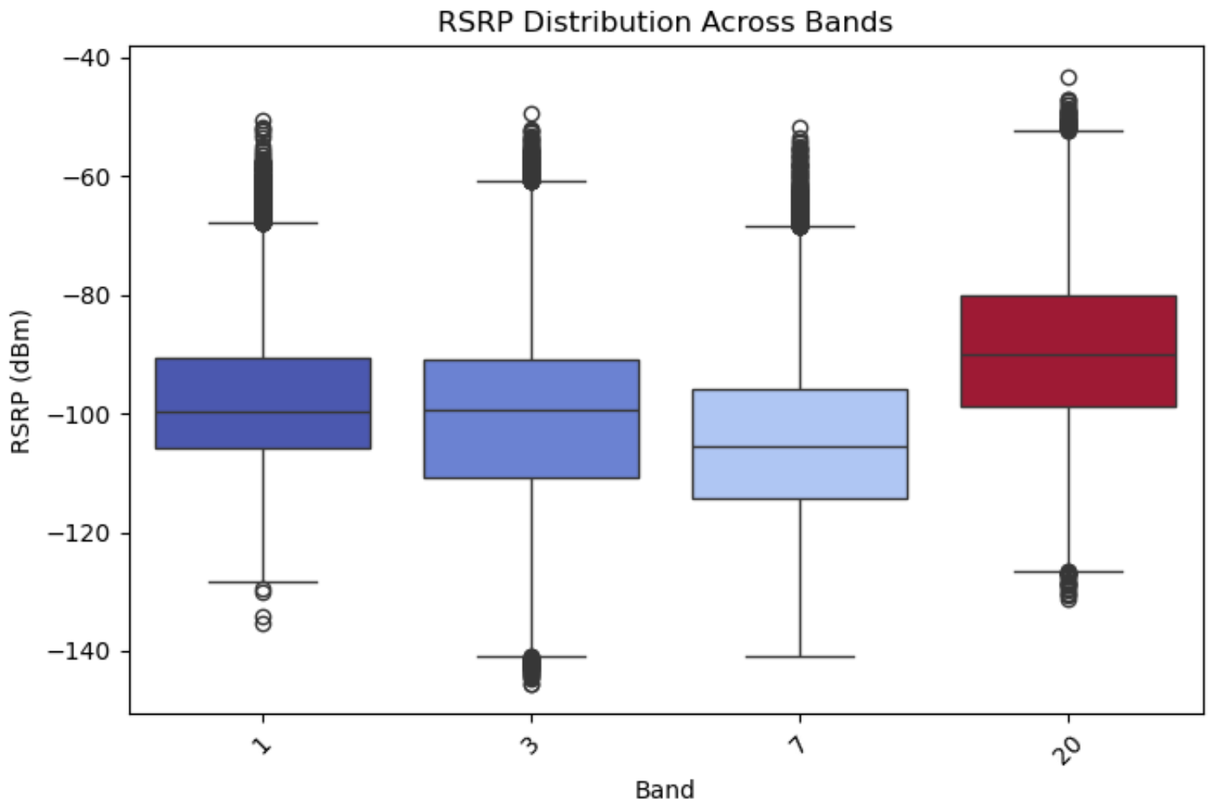


In [33]:
```
# Define the features to include in the pairplot
features = ['RSRP', 'Speed', 'cellPosErrorLambda1', 'cellPosErrorLambda2']

# Create the pairplot
sns.pairplot(df[features], diag_kind='kde', plot_kws={'alpha': 0.5})

# Set the title for the entire plot
plt.suptitle('Pairplot of Selected Features', y=1.02)  # y=1.02 adjusts the title position

# Display the plot
plt.show()
```

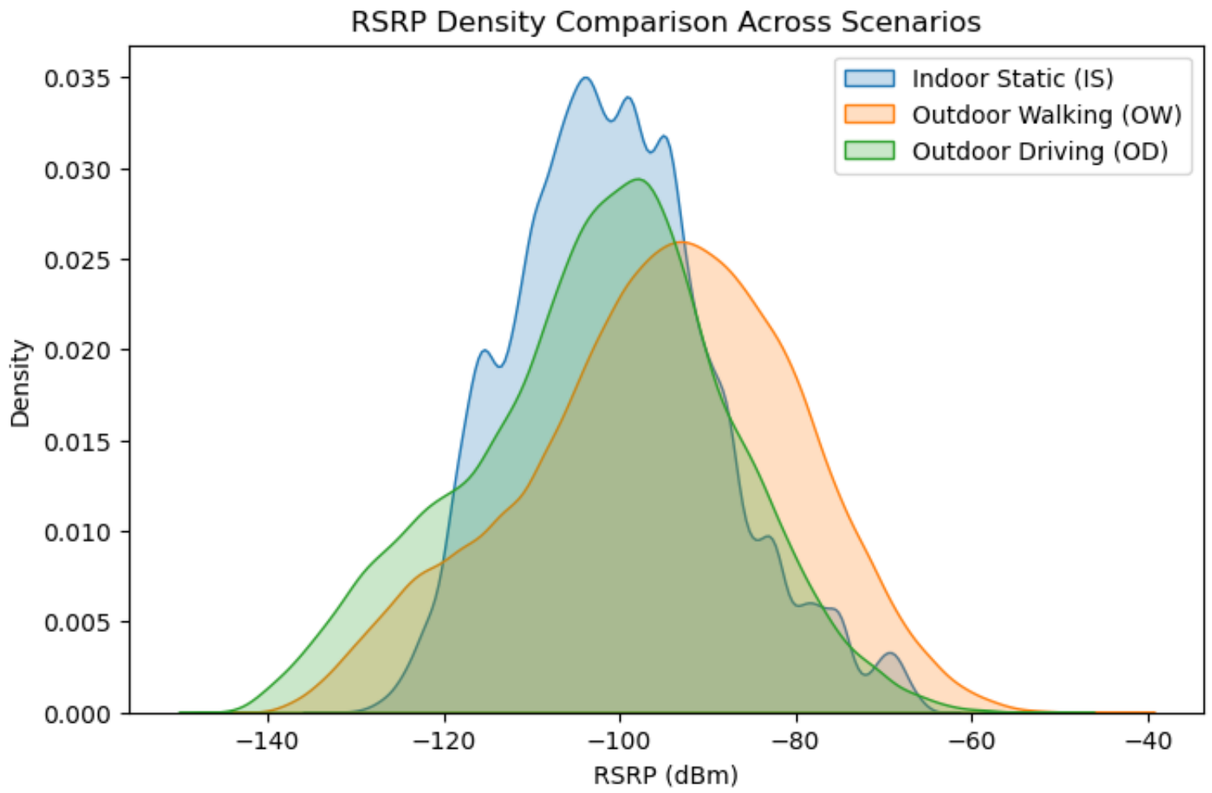




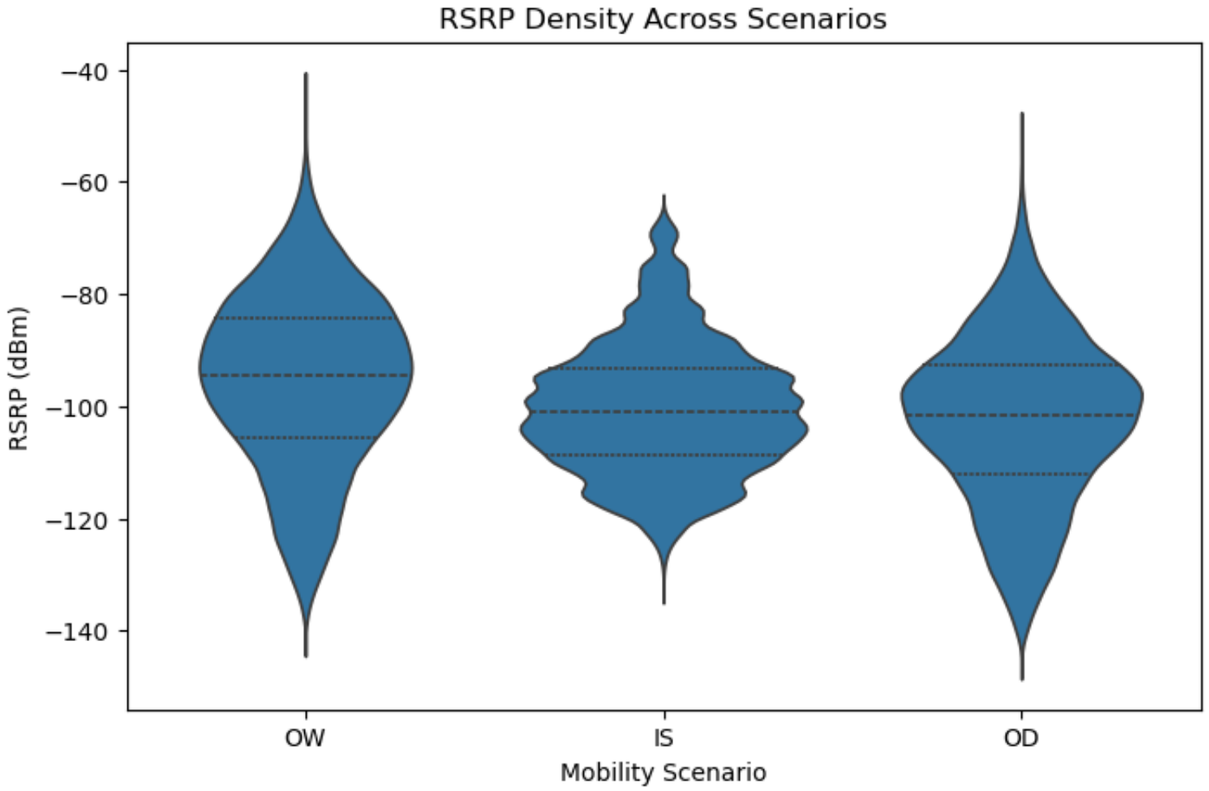
In [34]:
```
# RSRP Across Bands
plt.figure(figsize=(8, 5))
sns.boxplot(data=df, x="Band", y="RSRP", hue="Band", dodge=False, palette="coolwarm")
plt.title("RSRP Distribution Across Bands")
plt.xlabel("Band")
plt.ylabel("RSRP (dBm)")
plt.xticks(rotation=45)
plt.legend([],[], frameon=False)  # Removing redundant legend
plt.show()
```



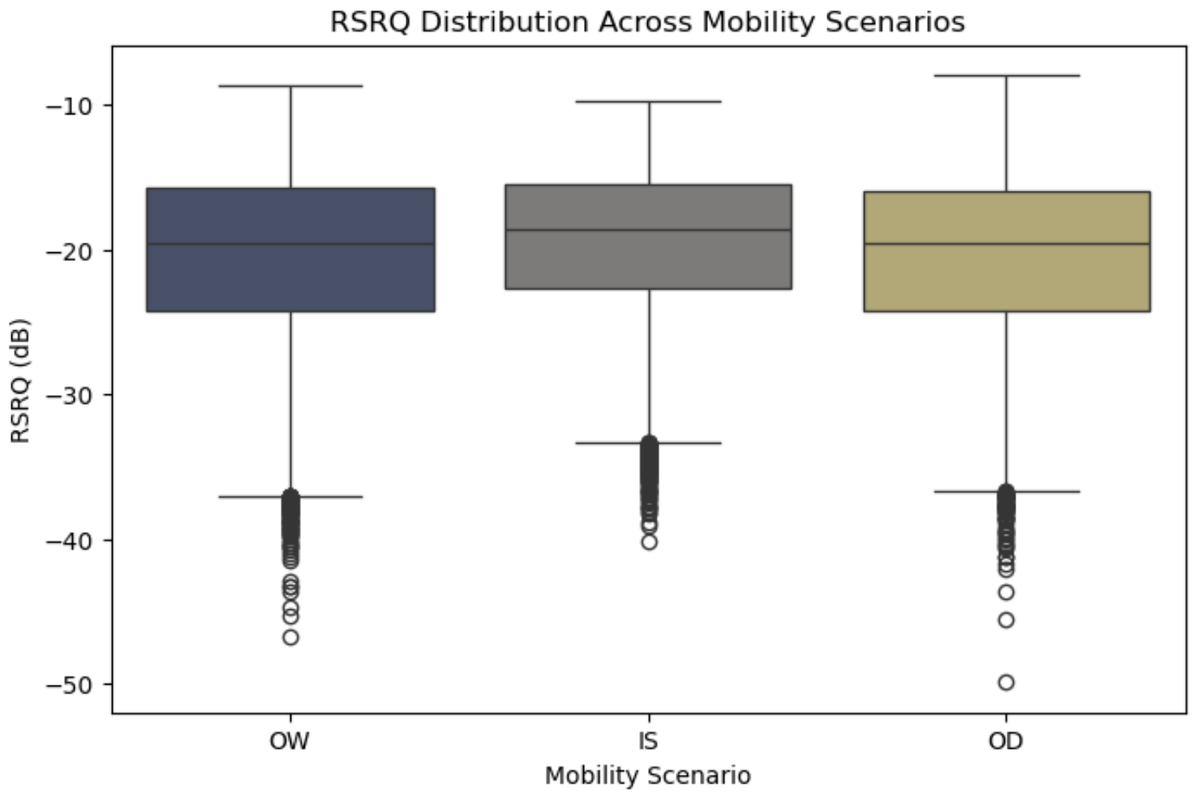
In [35]:
```
plt.figure(figsize=(8, 5))
sns.kdeplot(df[df['scenario'] == 'IS']['RSRP'], label='Indoor Static (IS)', fill=True)
sns.kdeplot(df[df['scenario'] == 'OW']['RSRP'], label='Outdoor Walking (OW)', fill=True)
sns.kdeplot(df[df['scenario'] == 'OD']['RSRP'], label='Outdoor Driving (OD)', fill=True)
plt.title("RSRP Density Comparison Across Scenarios")
plt.xlabel("RSRP (dBm)")
plt.ylabel("Density")
plt.legend()
plt.show()
```

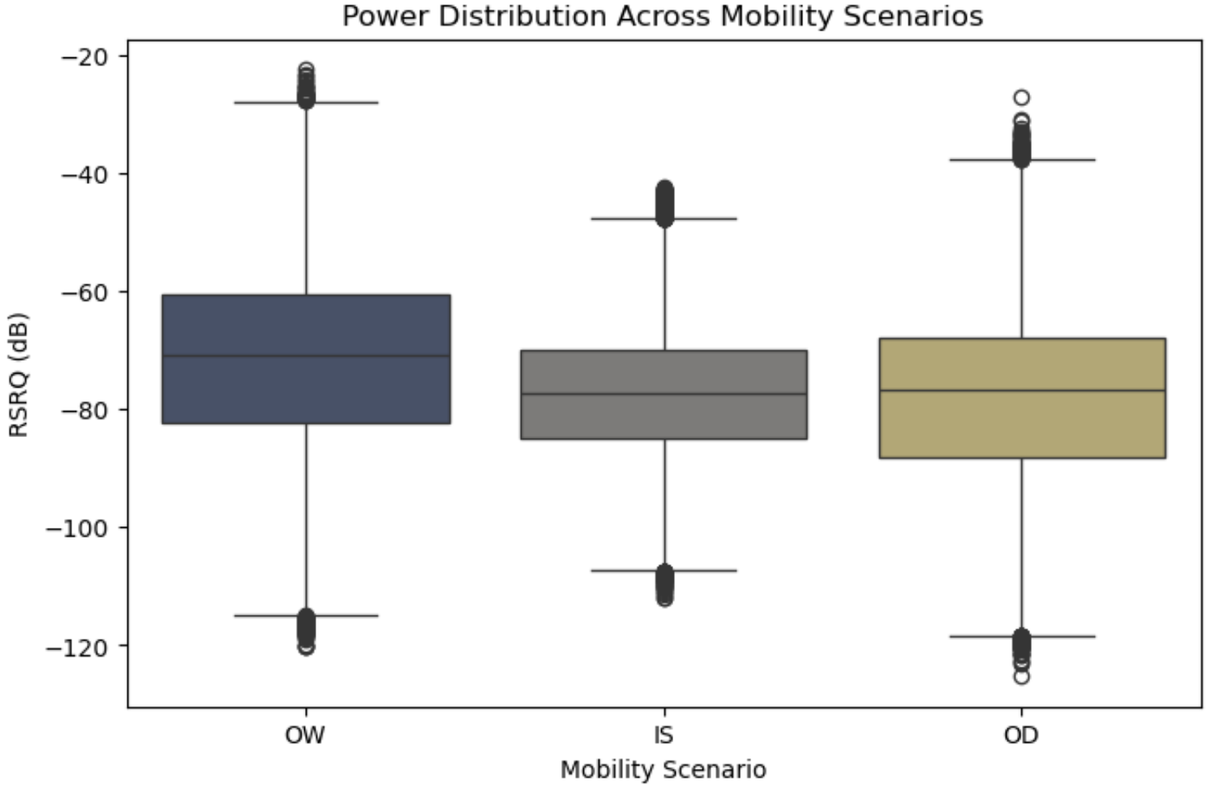


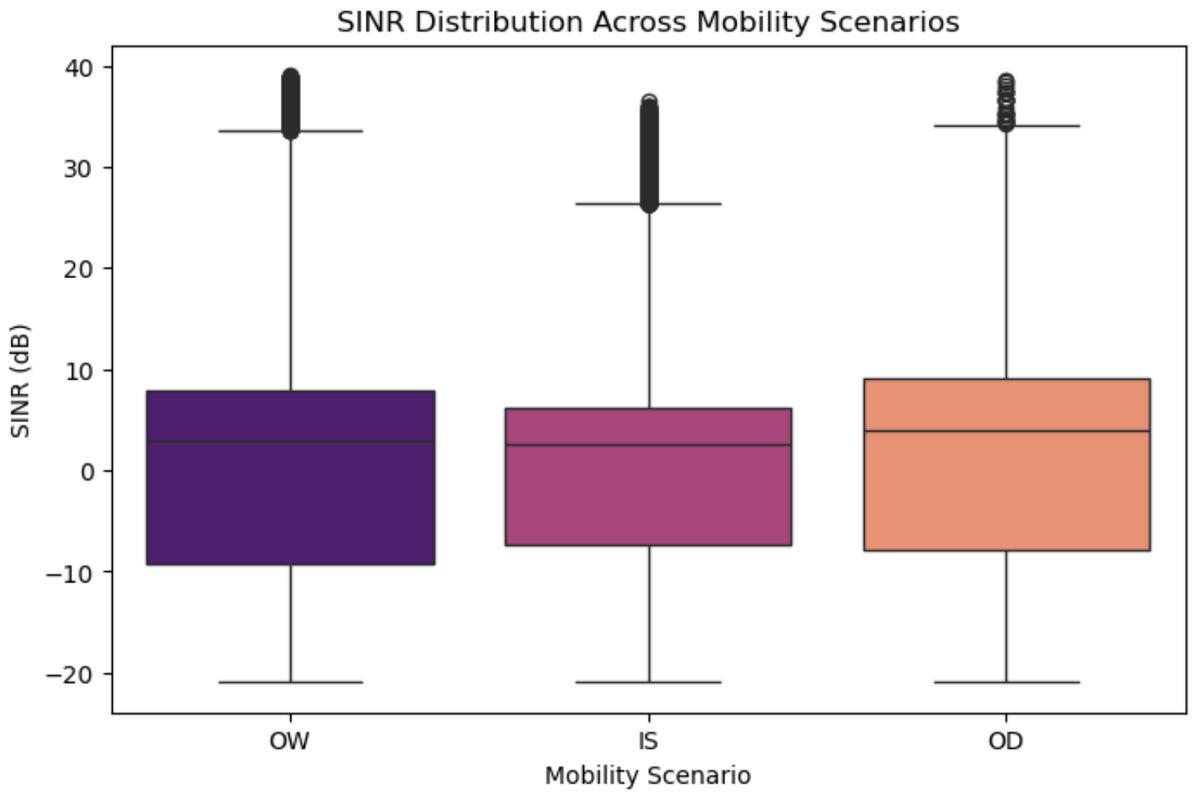
In [36]:
```
plt.figure(figsize=(8, 5))
sns.violinplot(x='scenario', y='RSRP', data=df, inner='quartile')
plt.title("RSRP Density Across Scenarios")
plt.xlabel("Mobility Scenario")
plt.ylabel("RSRP (dBm)")
plt.show()
```



In [37]:
```
# RSRQ Across Scenarios
plt.figure(figsize=(8, 5))
sns.boxplot(data=df, x="scenario", y="RSRQ", hue="scenario", dodge=False, palette="cividis")
plt.title("RSRQ Distribution Across Mobility Scenarios")
plt.xlabel("Mobility Scenario")
plt.ylabel("RSRQ (dB)")
plt.legend([],[], frameon=False)  # Remove redundant legend
plt.show()

# Power Across Scenarios
plt.figure(figsize=(8, 5))
sns.boxplot(data=df, x="scenario", y="Power", hue="scenario", dodge=False, palette="cividis")
plt.title("Power Distribution Across Mobility Scenarios")
plt.xlabel("Mobility Scenario")
plt.ylabel("RSRQ (dB)")
plt.legend([],[], frameon=False)  # Remove redundant legend
plt.show()

# SINR Across Scenarios
plt.figure(figsize=(8, 5))
sns.boxplot(data=df, x="scenario", y="SINR", hue="scenario", dodge=False, palette="magma")
plt.title("SINR Distribution Across Mobility Scenarios")
plt.xlabel("Mobility Scenario")
plt.ylabel("SINR (dB)")
plt.legend([],[], frameon=False)  # Remove redundant legend
plt.show()
```


RSRQ Distribution Across Mobility Scenarios
RSRQ (dB)
−10
−20
−30
−40
−50
OW
IS
OD
Mobility Scenario


Power Distribution Across Mobility Scenarios
RSRQ (dB)
−20
−40
−60
−80
−100
−120
OW
IS
OD
Mobility Scenario



In [38]:
```
# Correlation Analysis
correlation, p_value = stats.pearsonr(df["Speed"], df["RSRP"])

# Scenario-Based Analysis
plt.figure(figsize=(8, 5))
sns.scatterplot(data=df, x="Speed", y="RSRP", hue="scenario", alpha=0.3, palette="deep")
plt.title("RSRP vs. Speed Across Scenarios")
plt.xlabel("Speed (km/h)")
plt.ylabel("RSRP (dBm)")
plt.legend(title="Scenario")
plt.show()

# Display the results
correlation, p_value
```

Out[38]: (-0.0651243350595911, 0.0)

In [39]:
```
# Check unique values and their counts in the 'MNC', 'scenario', and 'campaign' columns before encoding
unique_mnc = df['MNC'].unique()
unique_scenario = df['scenario'].unique()
unique_campaign = df['campaign'].unique()

# Create a DataFrame to hold the unique values and their counts
data = {
    'Feature': ['MNC', 'scenario', 'campaign'],
    'Unique Values': [unique_mnc, unique_scenario, unique_campaign],
    'Count of Unique Values': [len(unique_mnc), len(unique_scenario), len(unique_campaign)],
}

df_unique = pd.DataFrame(data)

# Display the DataFrame
print(df_unique)
```

```
    Feature                                      Unique Values  \
0       MNC                                ["Op"[1], "Op"[2]]
1  scenario                                          [OW, IS, OD]
2  campaign  [campaign_6_OW_4G_gaming, campaign_35_IS_4G_sp...

   Count of Unique Values
0                       2
1                       3
2                     193
```

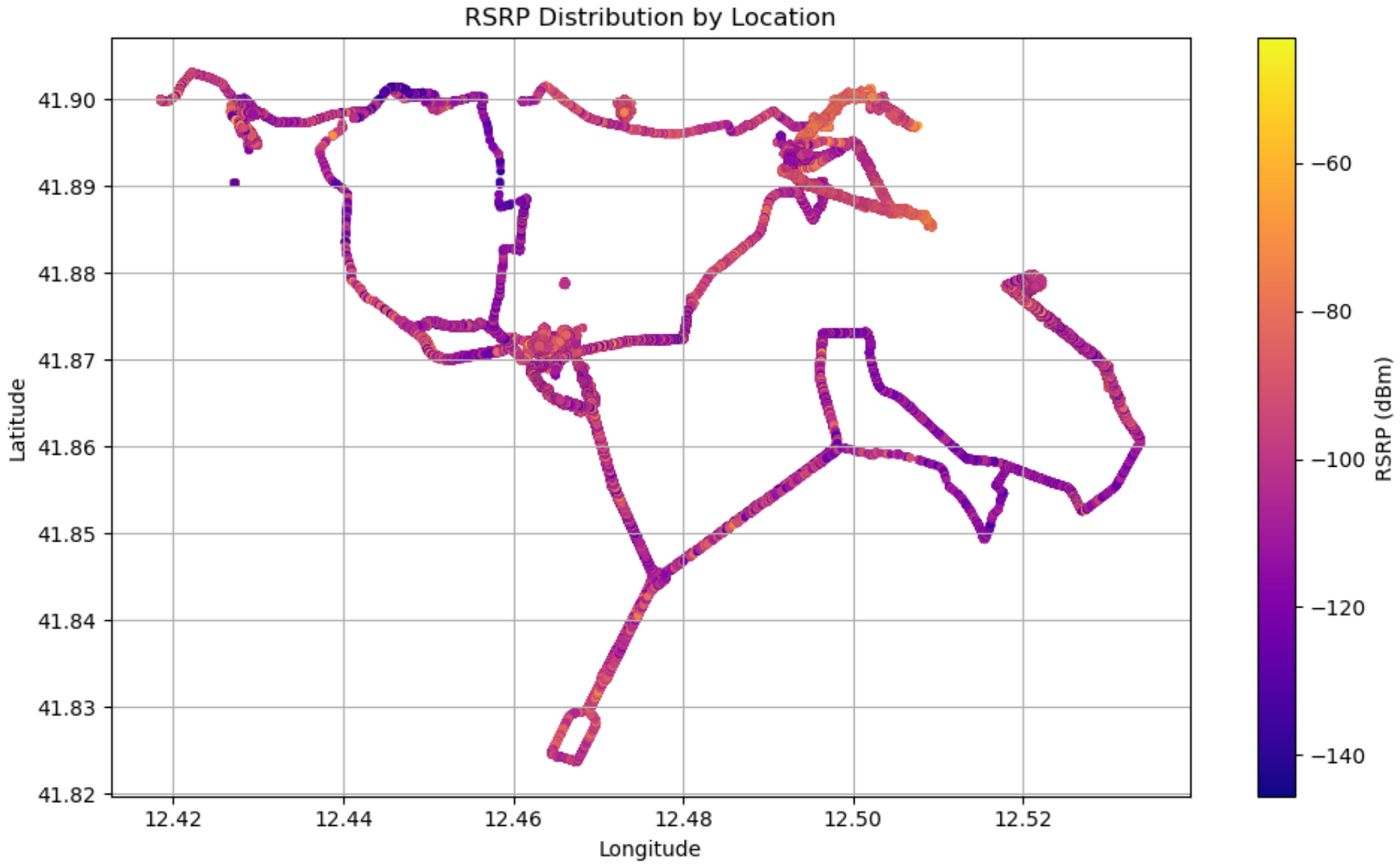
Encoding Categorical Features

In [41]:
```
# Create a copy of the original dataset
df_encoded = df.copy()
# One-Hot Encoding for 'MNC' and 'scenario'
df_encoded = pd.get_dummies(df_encoded, columns=['MNC', 'scenario'], drop_first=True)
# Convert specific one-hot encoded columns to integers
for col in df_encoded.columns:
    if col.startswith('MNC_') or col.startswith('scenario_'):
        df_encoded[col] = df_encoded[col].astype(int)
# Encoding 'campaign'
campaign_unique = df_encoded['campaign'].nunique()
print("Number of unique campaigns:", campaign_unique)

if campaign_unique < 10:
    df_encoded = pd.get_dummies(df_encoded, columns=['campaign'], drop_first=True)
else:
    le_campaign = LabelEncoder()
    df_encoded['campaign_encoded'] = le_campaign.fit_transform(df_encoded['campaign'])
    df_encoded.drop(columns=['campaign'], inplace=True)
# Process Date and Time Columns
df_encoded['Date'] = pd.to_datetime(df_encoded['Date'], format='%d.%m.%Y', errors='coerce')
df_encoded['Time'] = pd.to_datetime(df_encoded['Time'], format='%H:%M:%S.%f', errors='coerce').dt.time
# Check the resulting DataFrame
print(df_encoded.head())
```

```
Number of unique campaigns: 193
        Date             Time           UTC   Latitude  Longitude  Altitude  \
0 2021-01-14  09:19:28.214000  1.613291e+09  41.896722  12.507302     53.66
1 2021-01-14  09:19:28.214000  1.613291e+09  41.896722  12.507302     53.66
2 2021-01-14  09:19:28.840000  1.613291e+09  41.896721  12.507287     54.85
3 2021-01-14  09:19:36.195000  1.613291e+09  41.896759  12.507209     54.82
4 2021-01-14  09:19:36.549000  1.613291e+09  41.896759  12.507209     54.82

   Speed  EARFCN  Frequency  PCI  ...  cellLatitude  cellPosErrorLambda1  \
0   4.03    6300      806.0  412  ...     41.890300            10.610001
1   4.03    6300      806.0  411  ...     41.890300            10.610001
2   4.07    1350     1820.0  272  ...     41.890300             2.392951
3   4.03    3025     2647.5  266  ...     41.896951            12.490001
4   4.03    6300      806.0  412  ...     41.890300            10.610001

   cellPosErrorLambda2  n_CellIdentities    distance  Band  MNC_"Op"[2]  \
0            10.610001                 6  757.483987    20            0
1            10.610001                 6  757.483987    20            0
2             0.838046                 6  756.968865     3            0
3            12.490001                 6   21.962869     7            1
4            10.610001                 6  758.872653    20            0

   scenario_OD  scenario_OW  campaign_encoded
0            0            1               153
1            0            1               153
2            0            1               153
3            0            1               153
4            0            1               153

[5 rows x 27 columns]
```

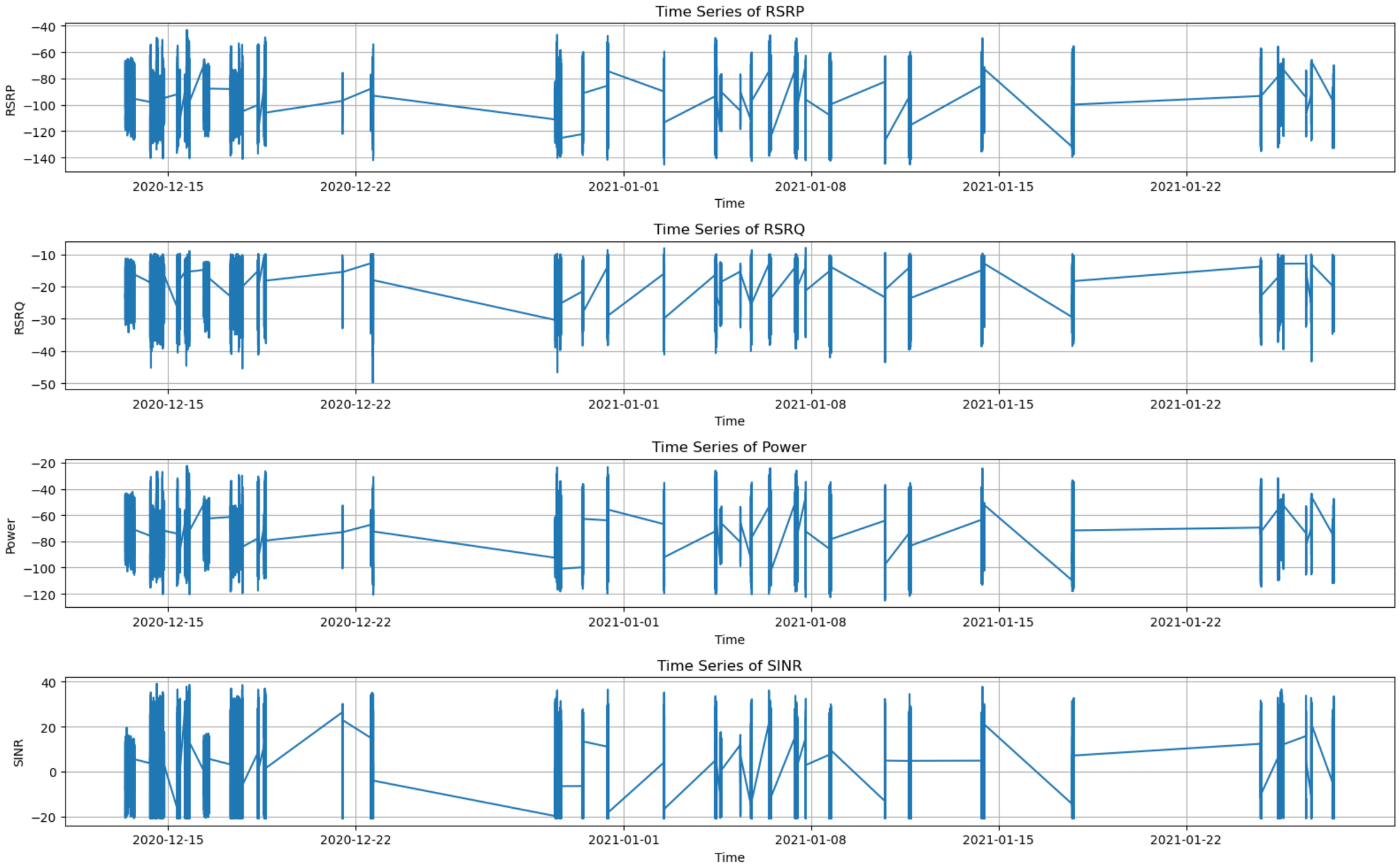
In [43]:

```
plt.figure(figsize=(10, 6))
scatter = plt.scatter(
    x=df['Longitude'],
    y=df['Latitude'],
    c=df['RSRP'],
    cmap='plasma',
    s=10
)
plt.colorbar(scatter, label='RSRP (dBm)')
plt.title('RSRP Distribution by Location')
plt.xlabel('Longitude')
plt.ylabel('Latitude')
plt.grid(True)
plt.tight_layout()
plt.show()
```



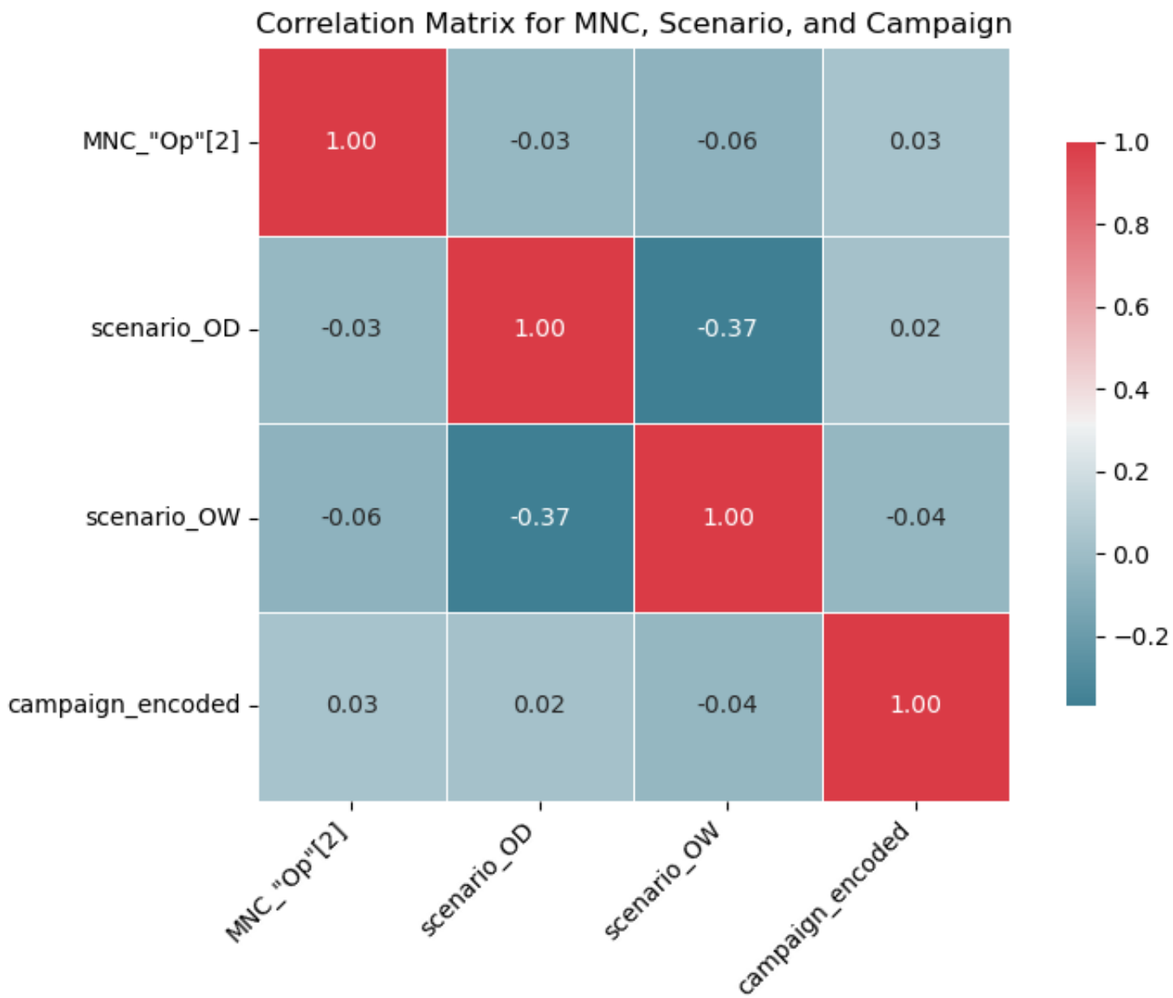
In [42]:
```
# Combine Date and Time into one datetime column
df_encoded['Datetime'] = pd.to_datetime(df_encoded['Date'].astype(str) + ' ' + df_encoded['Time'].astype(str), errors='coerce')

# Drop rows with invalid datetime
df_encoded = df_encoded.dropna(subset=['Datetime'])

# Set Datetime as index
df_encoded.set_index('Datetime', inplace=True)

# Sort by datetime index
df_encoded.sort_index(inplace=True)

# Plot time series for RSRP, RSRQ, Power, and SINR
plt.figure(figsize=(16, 10))
for i, metric in enumerate(['RSRP', 'RSRQ', 'Power', 'SINR'], 1):
    plt.subplot(4, 1, i)
    plt.plot(df_encoded[metric], label=metric)
    plt.title(f'Time Series of {metric}')
    plt.xlabel('Time')
    plt.ylabel(metric)
```

```
    plt.grid(True)
    plt.tight_layout()
plt.show()
```



In [44]:
```
# Select the one-hot encoded columns related to 'MNC', 'scenario', and 'campaign_encoded'
selected_columns = [col for col in df_encoded.columns if col.startswith('MNC') or col.startswith('scenario') or col == 'campaign_encoded']
# Subset the DataFrame with only the selected columns
df_selected = df_encoded[selected_columns].copy()
```

```
# Calculate correlations between these selected features
corrmat = df_selected.corr()
# Set up the matplotlib figure
fig, ax = plt.subplots(figsize=(8, 6))  # Adjust the size as needed
# Generate a custom diverging colormap
cmap = sns.diverging_palette(220, 10, as_cmap=True)
# Draw the heatmap with the correlation matrix
sns.heatmap(corrmat, cmap=cmap, annot=True, fmt='.2f', linewidths=0.5,
            cbar_kws={"shrink": 0.75}, annot_kws={"size": 10},  # Adjust annotation font size
            square=True, ax=ax)
# Rotate the tick labels for better readability
plt.xticks(rotation=45, ha='right', fontsize=10)
plt.yticks(rotation=0, fontsize=10)
# Set the title with appropriate font size
plt.title('Correlation Matrix for MNC, Scenario, and Campaign', fontsize=12)
# Adjust layout to ensure everything fits without overlap
plt.tight_layout()
# Display the heatmap
plt.show()
```



In [45]:
```
# Saves a copy of the encoded Dataframe
```

```
df_transformed = df_encoded.copy()
```

Create a Sample of the Dataset

In [47]:
```
# Create a sample of 40,000 records from the cleaned DataFrame
df_sample = df_cleaned.sample(n=40000, random_state=42)

# Verify the sample size
print("Sample shape:", df_sample.shape)
df_sample.info()
```

```
Sample shape: (40000, 26)
<class 'pandas.core.frame.DataFrame'>
Index: 40000 entries, 429425 to 137755
Data columns (total 26 columns):
 #   Column               Non-Null Count  Dtype
---  ------               --------------  -----
 0   Date                 40000 non-null  object
 1   Time                 40000 non-null  object
 2   UTC                  40000 non-null  float64
 3   Latitude             40000 non-null  float64
 4   Longitude            40000 non-null  float64
 5   Altitude             40000 non-null  float64
 6   Speed                40000 non-null  float64
 7   EARFCN               40000 non-null  int64
 8   Frequency            40000 non-null  float64
 9   PCI                  40000 non-null  int64
 10  MNC                  40000 non-null  object
 11  CellIdentity         40000 non-null  int64
 12  eNodeB.ID            40000 non-null  int64
 13  Power                40000 non-null  float64
 14  SINR                 40000 non-null  float64
 15  RSRP                 40000 non-null  float64
 16  RSRQ                 40000 non-null  float64
 17  scenario             40000 non-null  object
 18  cellLongitude        40000 non-null  float64
 19  cellLatitude         40000 non-null  float64
 20  cellPosErrorLambda1  40000 non-null  float64
 21  cellPosErrorLambda2  40000 non-null  float64
 22  n_CellIdentities     40000 non-null  int64
 23  distance             40000 non-null  float64
 24  Band                 40000 non-null  int64
 25  campaign             40000 non-null  object
dtypes: float64(15), int64(6), object(5)
memory usage: 8.2+ MB
```

In [48]:
```
df_sample.to_csv('sampled_data.csv', index=False)
```

In [49]:
```
# Summary statistics
print("\nSummary Statistics:")
print(df_sample[["distance", "RSRP", "Band", "Speed"]].describe())
```

```
Summary Statistics:
          distance          RSRP          Band         Speed
count  40000.000000  40000.000000  40000.000000  40000.000000
mean     552.548080    -98.560913      7.688675      5.178591
std      784.234602     14.150622      7.086840      9.430004
min        0.222639   -142.180000      1.000000      0.000000
25%      158.577997   -108.050000      3.000000      0.680000
50%      286.565321    -98.500000      3.000000      2.590000
75%      618.048584    -89.147500      7.000000      4.680000
max     8617.739658    -50.430000     20.000000     76.210000
```

Standardising Numerical Features

In [51]:
```
# Feature Scaling
sc = StandardScaler()
df_scaled = sc.fit_transform(df_sample[["distance", "RSRP", "Band", "Speed"]])
```

In [52]:
```
# Convert df_scaled to a DataFrame
df_scaled_df = pd.DataFrame(df_scaled, columns=df[["distance", "RSRP", "Band", "Speed"]].columns)
print("\nSummary Statistics:")
print(df_scaled_df.describe())
```

```
Summary Statistics:
          distance          RSRP          Band         Speed
count  4.000000e+04  4.000000e+04  4.000000e+04  4.000000e+04
mean  -6.892265e-17 -1.053380e-15 -1.705303e-17 -2.131628e-17
std    1.000013e+00  1.000013e+00  1.000013e+00  1.000013e+00
min   -7.042948e-01 -3.082524e+00 -9.438280e-01 -5.491680e-01
25%   -5.023688e-01 -6.705857e-01 -6.616113e-01 -4.770568e-01
50%   -3.391665e-01  4.304670e-03 -6.616113e-01 -2.745093e-01
75%    8.352261e-02  6.652379e-01 -9.717781e-02 -5.287352e-02
max    1.028429e+01  3.401371e+00  1.737231e+00  7.532584e+00
```

TASK 2

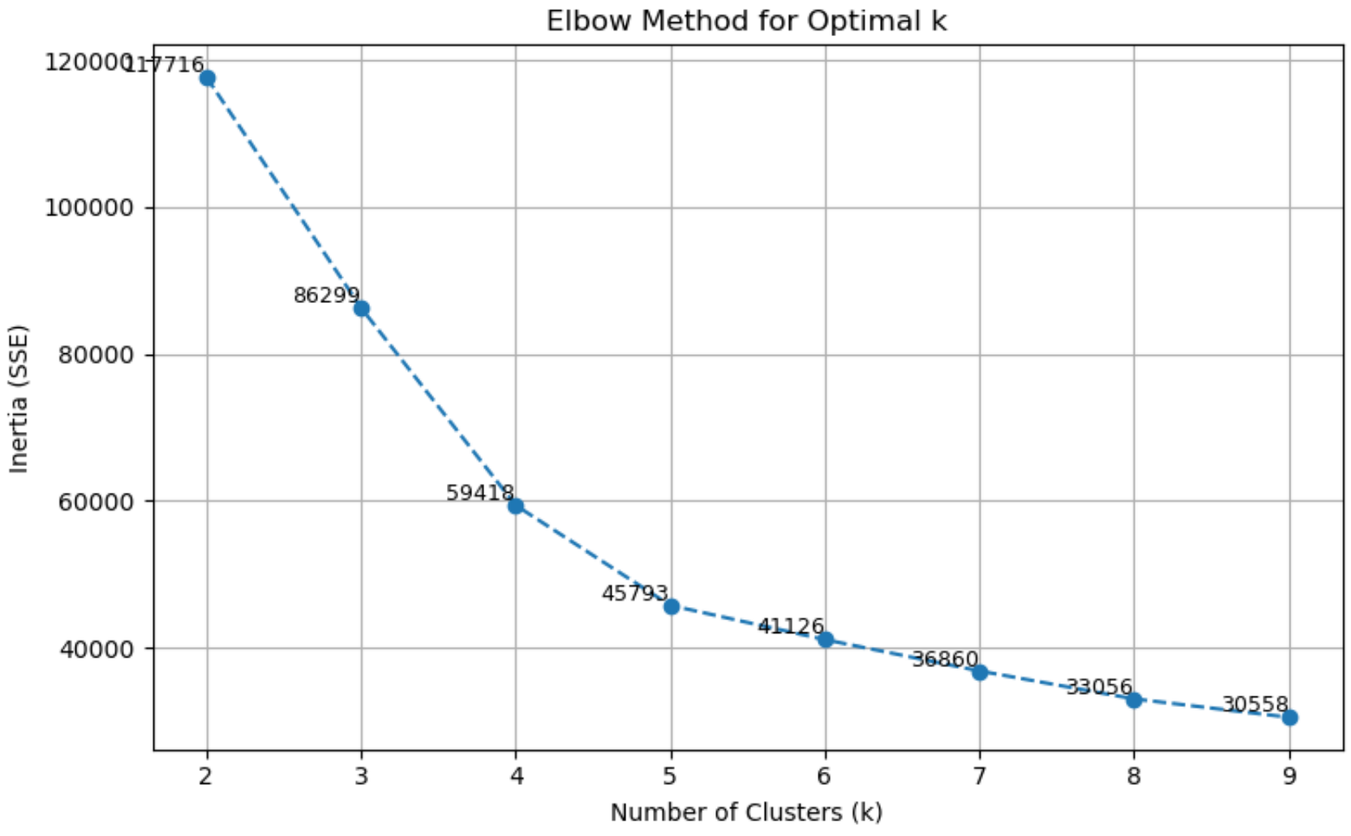
CLUSTERING

Analysis of LTE User Clustering in Relation to Signal Strength

KMeans

In [55]:
```
# Finding the optimal k using the Elbow Method
inertia = []
k_values = range(2, 10)

for k in k_values:
    kmeans = KMeans(n_clusters=k, random_state=42, n_init=10)
    kmeans.fit(df_scaled)
    inertia.append(kmeans.inertia_)
```

```
# Plot Elbow Method with annotated SSE values
plt.figure(figsize=(8, 5))
plt.plot(k_values, inertia, marker='o', linestyle='--')

# Annotate each point with its SSE value
for i, sse in enumerate(inertia):
    plt.text(k_values[i], sse, f"{sse:.0f}", ha='right', va='bottom', fontsize=9)

plt.xlabel("Number of Clusters (k)")
plt.ylabel("Inertia (SSE)")
plt.title("Elbow Method for Optimal k")
plt.grid(True)
plt.tight_layout()
plt.show()
```



In [56]:
```
# Select optimal k from elbow plot
optimal_k = 4
kmeans = KMeans(n_clusters=optimal_k, random_state=42, n_init=10)

# Fit K-Means and assign clusters
df_clustered = df_sample.copy()
df_clustered["KMeans_Cluster"] = kmeans.fit_predict(df_scaled)

# Get the cluster centers (in standardized form)
cluster_centers_scaled = kmeans.cluster_centers_
```

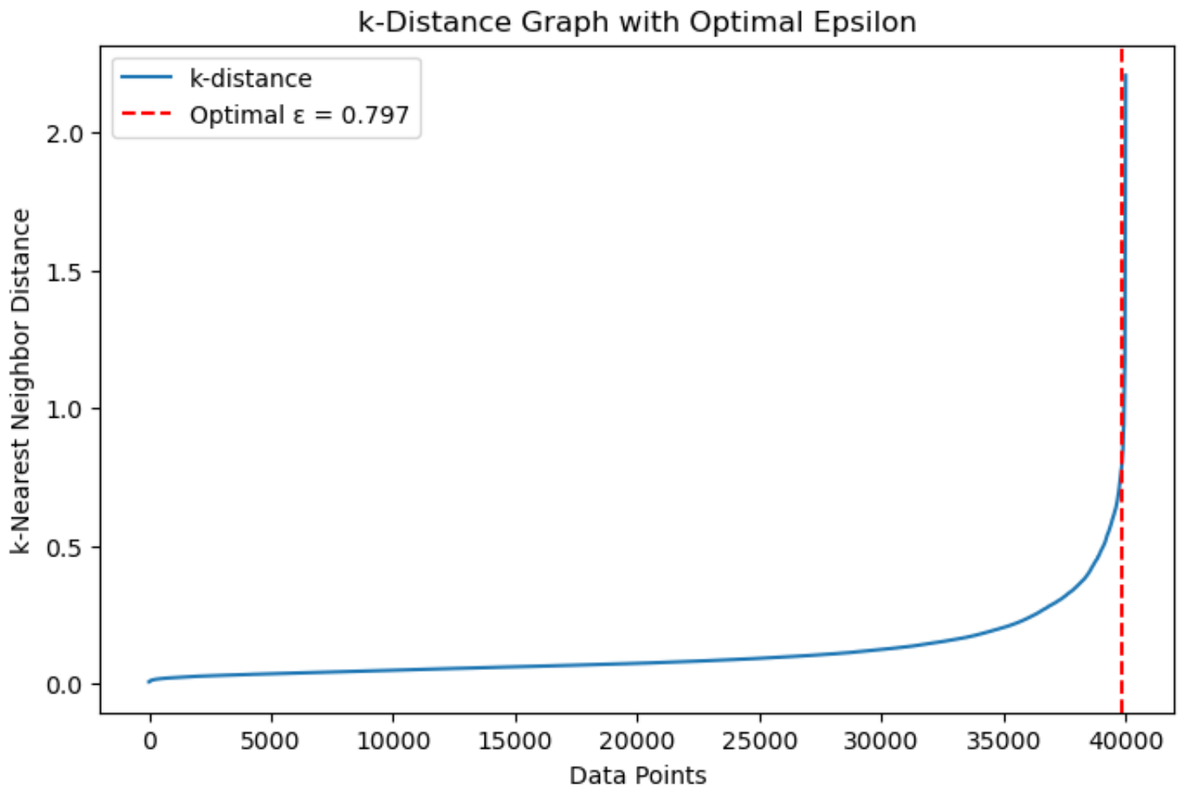
```
# Create a DataFrame for better readability using only the four features
columns = ["distance", "RSRP", "Band", "Speed"]
df_centers_scaled = pd.DataFrame(cluster_centers_scaled, columns=columns)

# Output cluster centers
print("Cluster Centers in Scaled Format:")
print(df_centers_scaled)
```

```
Cluster Centers in Scaled Format:
   distance      RSRP      Band     Speed
0 -0.255171 -0.068566 -0.520193 -0.233910
1 -0.143245  0.680149  1.736191 -0.189403
2  0.108305 -0.372270 -0.019368  3.580934
3  3.126508 -1.345892 -0.606509 -0.089897
```

In [ ]:

DBScan

In [58]:
```
# Adjust min_samples based on feature set
min_samples = 8

# Find nearest neighbors
neighbors = NearestNeighbors(n_neighbors=min_samples)
neighbors_fit = neighbors.fit(df_scaled)
distances, indices = neighbors_fit.kneighbors(df_scaled)

# Sort distances using the farthest nearest neighbor (last column)
distances = np.sort(distances[:, -1])  # Correct nearest neighbor distance

# Compute the knee (elbow point) numerically
kneedle = KneeLocator(np.arange(len(distances)), distances, curve="convex", direction="increasing")

# Get the optimal epsilon
optimal_eps = distances[kneedle.elbow]  # Use .elbow instead of .knee

# Plot with the knee point
plt.figure(figsize=(8, 5))
plt.plot(distances, label="k-distance")
plt.axvline(x=kneedle.elbow, color="r", linestyle="--", label=f"Optimal ε = {optimal_eps:.3f}")
plt.xlabel("Data Points")
plt.ylabel("k-Nearest Neighbor Distance")
plt.title("k-Distance Graph with Optimal Epsilon")
plt.legend()
plt.show()

# Display the optimal epsilon value
optimal_eps
```



Out[58]: 0.7966801264169214

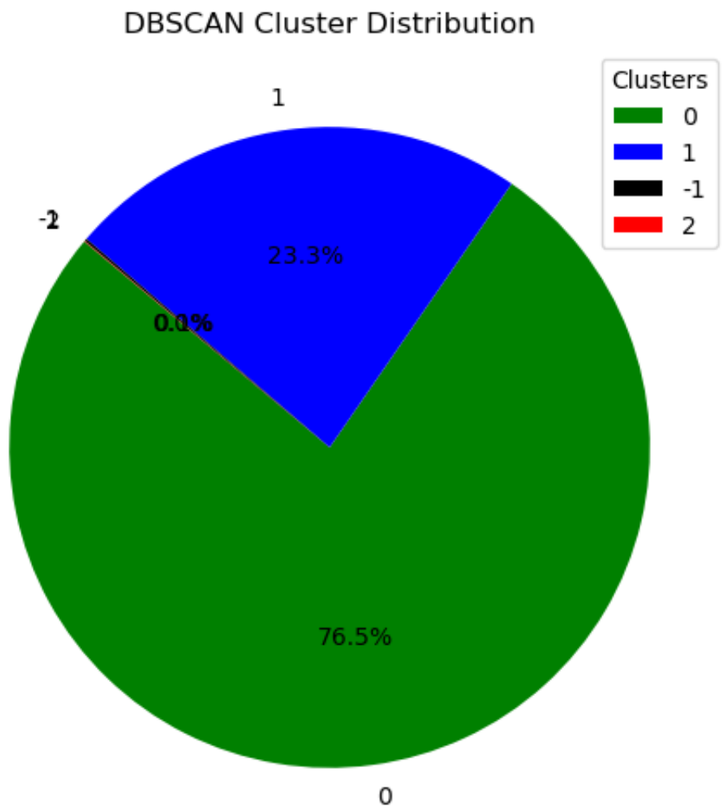
In [59]:
```
# Set optimal eps from the graph
dbscan = DBSCAN(eps=0.797, min_samples=8)
df_clustered["DBSCAN_Cluster"] = dbscan.fit_predict(df_scaled)
```

In [60]:
```
# Check the Number of Noise Points (-1 Labels)
print(df_clustered["DBSCAN_Cluster"].value_counts())
```

```
DBSCAN_Cluster
 0    30596
 1     9336
-1       55
 2       13
Name: count, dtype: int64
```

In [61]:
```
# Ensure `df_clustered` contains the cluster labels
if "DBSCAN_Cluster" not in df_clustered.columns:
    raise ValueError("DBSCAN_Cluster column is missing from df_clustered.")

# Count occurrences of each cluster
cluster_counts = df_clustered["DBSCAN_Cluster"].value_counts()

# Define cluster colors
cluster_colors = {
    -1: "black",  # Noise points (outliers)
    0: "green",
    1: "blue",
```

```
    2: "red",
}

# Assign colors based on cluster labels
colors = [cluster_colors.get(label, "gray") for label in cluster_counts.index]  # Default to gray if unknown

# Plot Pie Chart
plt.figure(figsize=(8, 6))
plt.pie(cluster_counts, labels=cluster_counts.index, autopct="%1.1f%%", startangle=140, colors=colors)
plt.title("DBSCAN Cluster Distribution")
plt.legend(title="Clusters", loc="upper right")
plt.show()
```



Silhouette Score

In [63]:
```
# Evaluate K-Means
silhouette_kmeans = silhouette_score(df_scaled, df_clustered["KMeans_Cluster"])
print(f"Silhouette Score (K-Means): {silhouette_kmeans:.4f}")

# Evaluate DBSCAN
if len(set(df_clustered["DBSCAN_Cluster"])) > 1:
    silhouette_dbscan = silhouette_score(df_scaled, df_clustered["DBSCAN_Cluster"])
    print(f"Silhouette Score (DBSCAN): {silhouette_dbscan:.4f}")
else:
```

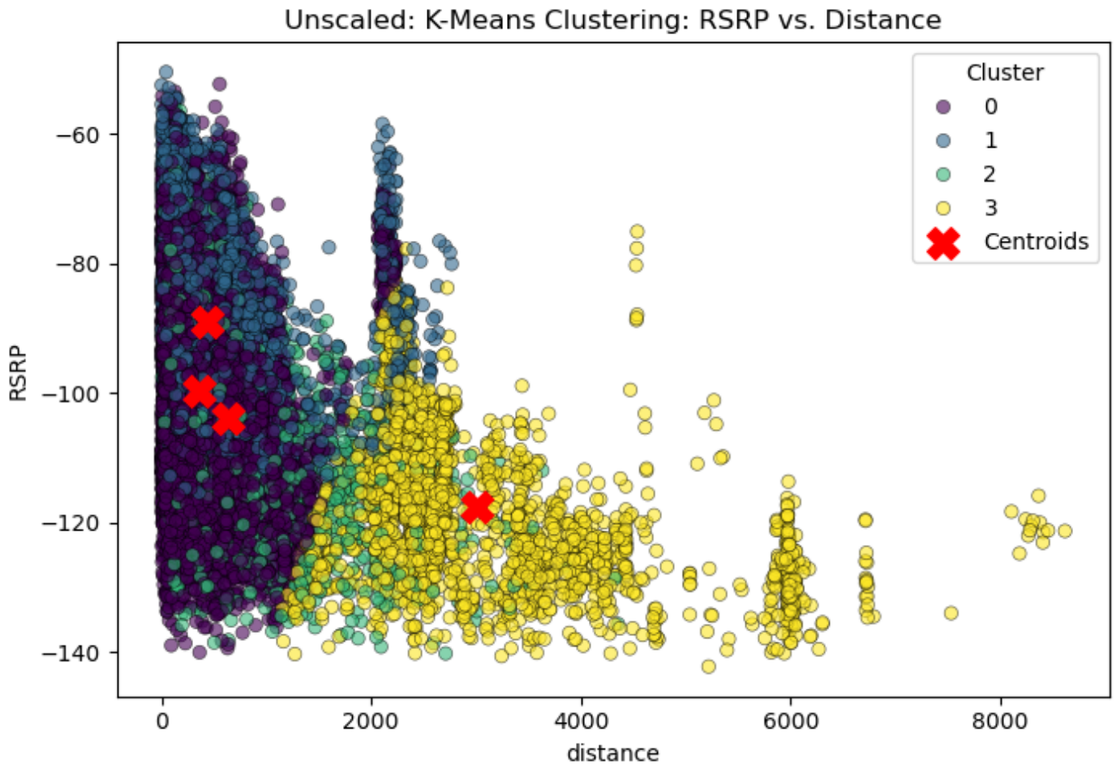
```
    print("DBSCAN only found one cluster or too much noise.")
```

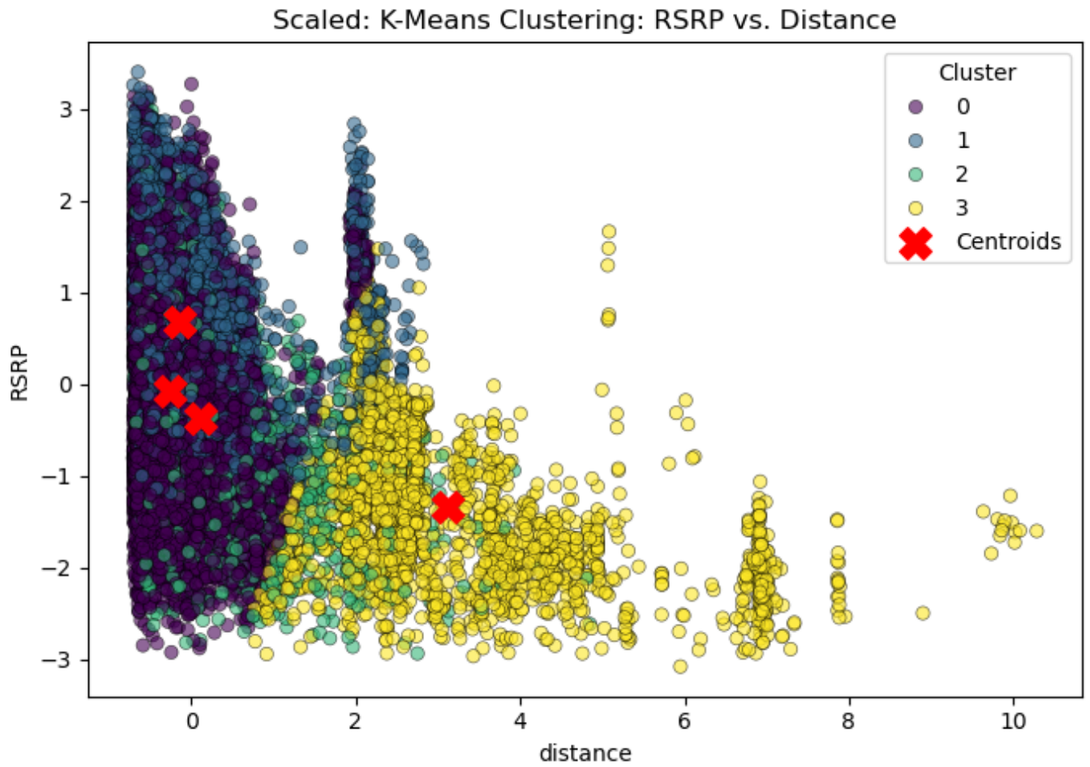
```
Silhouette Score (K-Means): 0.4910
Silhouette Score (DBSCAN): 0.3960
```

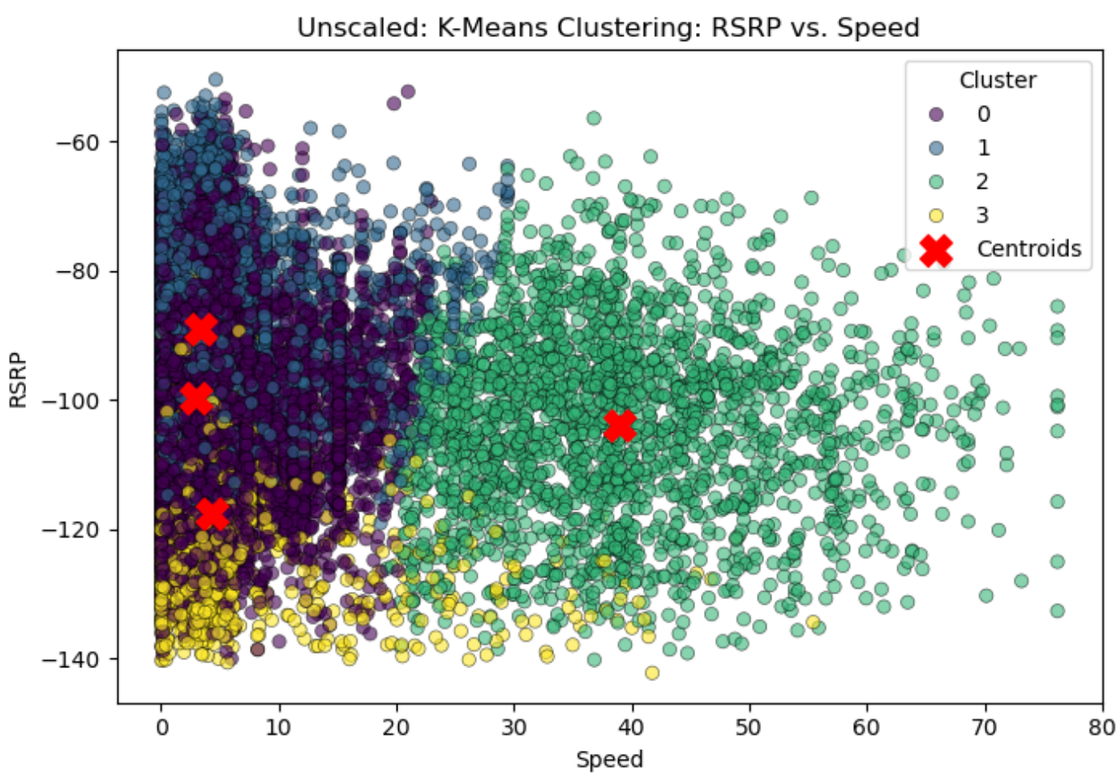
In [64]:
```
# df_clustered contains both numerical and categorical features
df_original = df_clustered.copy()
# Select numerical features  for scaling
numeric_features = ['distance', 'RSRP', 'Band', 'Speed']
scaler = StandardScaler()

# Scale only numerical features and convert back to DataFrame
scaled_array = scaler.fit_transform(df_original[numeric_features])
df_scaled = pd.DataFrame(scaled_array, columns=numeric_features, index=df_original.index)
```

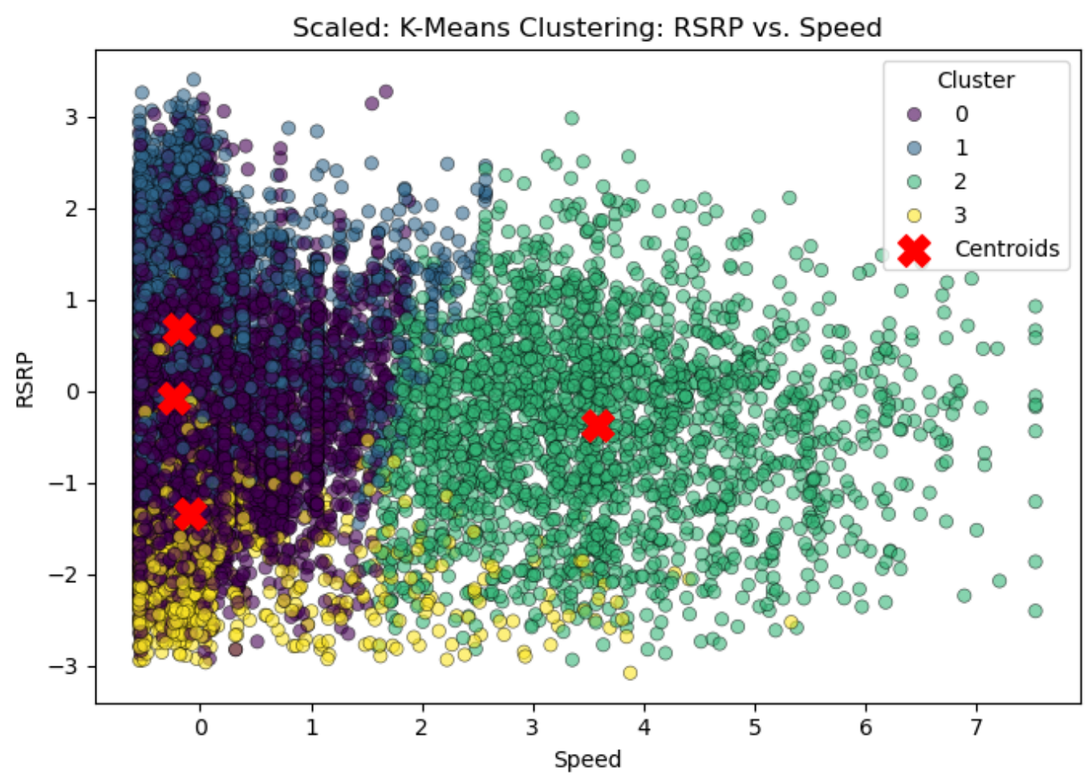
In [65]:
```
# Ensure the cluster labels are retained in the scaled dataset
df_scaled["KMeans_Cluster"] = df_original["KMeans_Cluster"]

# Convert centroids back to original feature space
centroids_scaled = kmeans.cluster_centers_
centroids_original = scaler.inverse_transform(centroids_scaled)  # Unscaled centroids

def plot_clusters_side_by_side(df_original, df_scaled, x_feature, y_feature, cluster_column, title, centroids_original=None, centroids_scaled=None):
    """
    Plots unscaled and scaled scatter plots side by side for better visualization,
    including centroids if provided.

    Parameters:
    df_original (DataFrame): The original (unscaled) dataset.
    df_scaled (DataFrame): The scaled dataset.
    x_feature (str): Feature for x-axis.
    y_feature (str): Feature for y-axis.
    cluster_column (str): Column representing clusters.
    title (str): Title for the plots.
    centroids_original (array): Centroid positions in the original scale.
    centroids_scaled (array): Centroid positions in the scaled space.
    """
    fig, axes = plt.subplots(1, 2, figsize=(14, 5))  # Side-by-side plots

    # Unscaled Scatter Plot
    sns.scatterplot(
        data=df_original, x=x_feature, y=y_feature, hue=cluster_column, palette='viridis', alpha=0.6, edgecolor='k', ax=axes[0]
    )
    if centroids_original is not None:
        axes[0].scatter(centroids_original[:, 0], centroids_original[:, 1], marker='X', s=200, color='red', label='Centroids')
    axes[0].set_title(f"Unscaled: {title}")
    axes[0].set_xlabel(x_feature)
    axes[0].set_ylabel(y_feature)
    axes[0].legend(title='Cluster')

    # Scaled Scatter Plot
    sns.scatterplot(
        data=df_scaled, x=x_feature, y=y_feature, hue=cluster_column, palette='viridis', alpha=0.6, edgecolor='k', ax=axes[1]
    )
    if centroids_scaled is not None:
        axes[1].scatter(centroids_scaled[:, 0], centroids_scaled[:, 1], marker='X', s=200, color='red', label='Centroids')
```

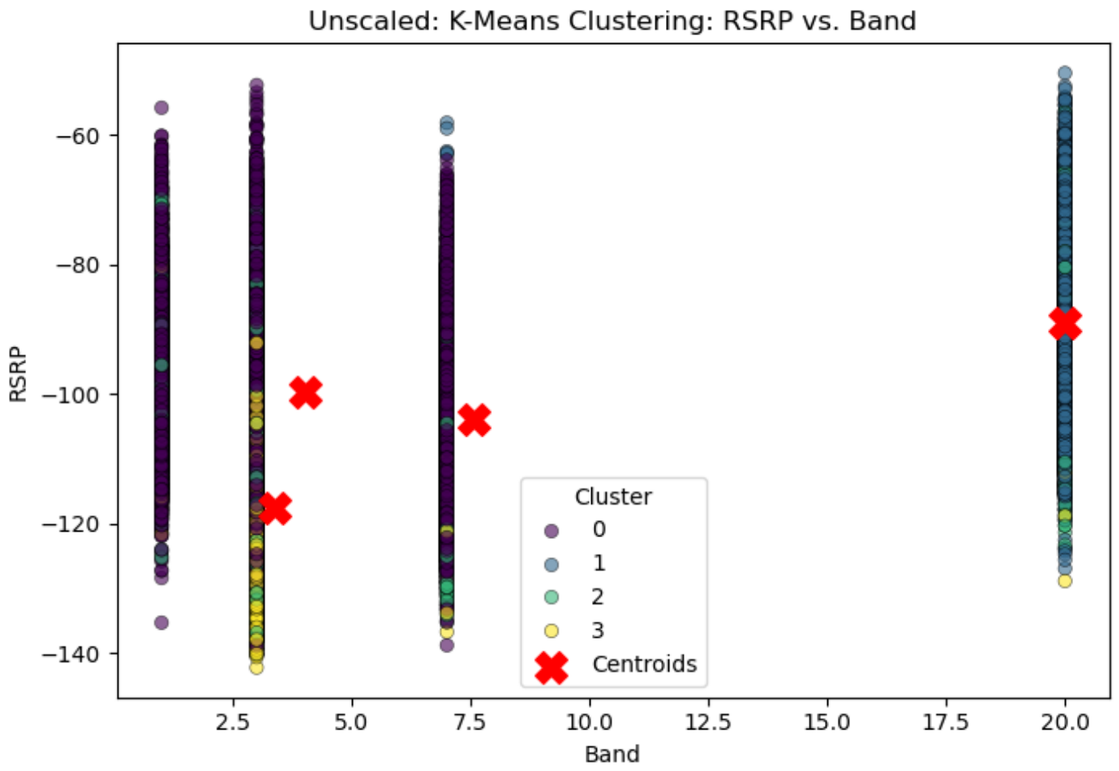
```
    axes[1].set_title(f"Scaled: {title}")
    axes[1].set_xlabel(x_feature)
    axes[1].set_ylabel(y_feature)
    axes[1].legend(title='Cluster')

    plt.tight_layout()
    plt.show()

# Feature index mapping for centroids
distance_idx, rsrp_idx = 0, 1
speed_idx, band_idx = 3, 2

# Extract centroid positions for each plot
centroids_plot_distance = centroids_original[:, [distance_idx, rsrp_idx]]
centroids_scaled_distance = centroids_scaled[:, [distance_idx, rsrp_idx]]

centroids_plot_speed = centroids_original[:, [speed_idx, rsrp_idx]]
centroids_scaled_speed = centroids_scaled[:, [speed_idx, rsrp_idx]]

centroids_plot_band = centroids_original[:, [band_idx, rsrp_idx]]
centroids_scaled_band = centroids_scaled[:, [band_idx, rsrp_idx]]

# Plot Side-by-Side Comparisons for Each Feature Pair
plot_clusters_side_by_side(df_original, df_scaled, 'distance', 'RSRP', 'KMeans_Cluster',
                           'K-Means Clustering: RSRP vs. Distance', centroids_plot_distance, centroids_scaled_distance)

plot_clusters_side_by_side(df_original, df_scaled, 'Speed', 'RSRP', 'KMeans_Cluster',
                           'K-Means Clustering: RSRP vs. Speed', centroids_plot_speed, centroids_scaled_speed)

plot_clusters_side_by_side(df_original, df_scaled, 'Band', 'RSRP', 'KMeans_Cluster',
                           'K-Means Clustering: RSRP vs. Band', centroids_plot_band, centroids_scaled_band)
```

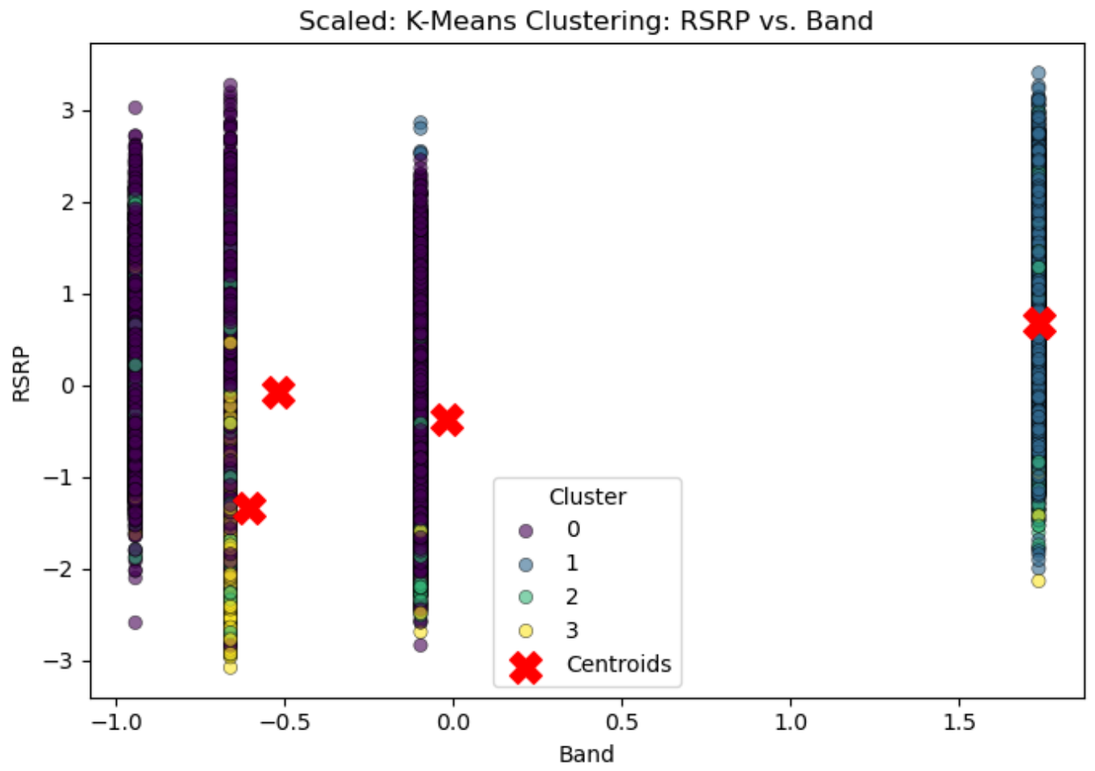

Unscaled: K-Means Clustering: RSRP vs. Distance
RSRP
distance
Cluster
0
1
2
3
Centroids
−60
−80
−100
−120
−140
0
2000
4000
6000
8000


Scaled: K-Means Clustering: RSRP vs. Distance
RSRP
distance
Cluster
0
1
2
3
Centroids
3
2
1
0
−1
−2
−3
0
2
4
6
8
10


Unscaled: K-Means Clustering: RSRP vs. Speed
RSRP
Speed
Cluster
0
1
2
3
Centroids
−60
−80
−100
−120
−140
0
10
20
30
40
50
60
70
80

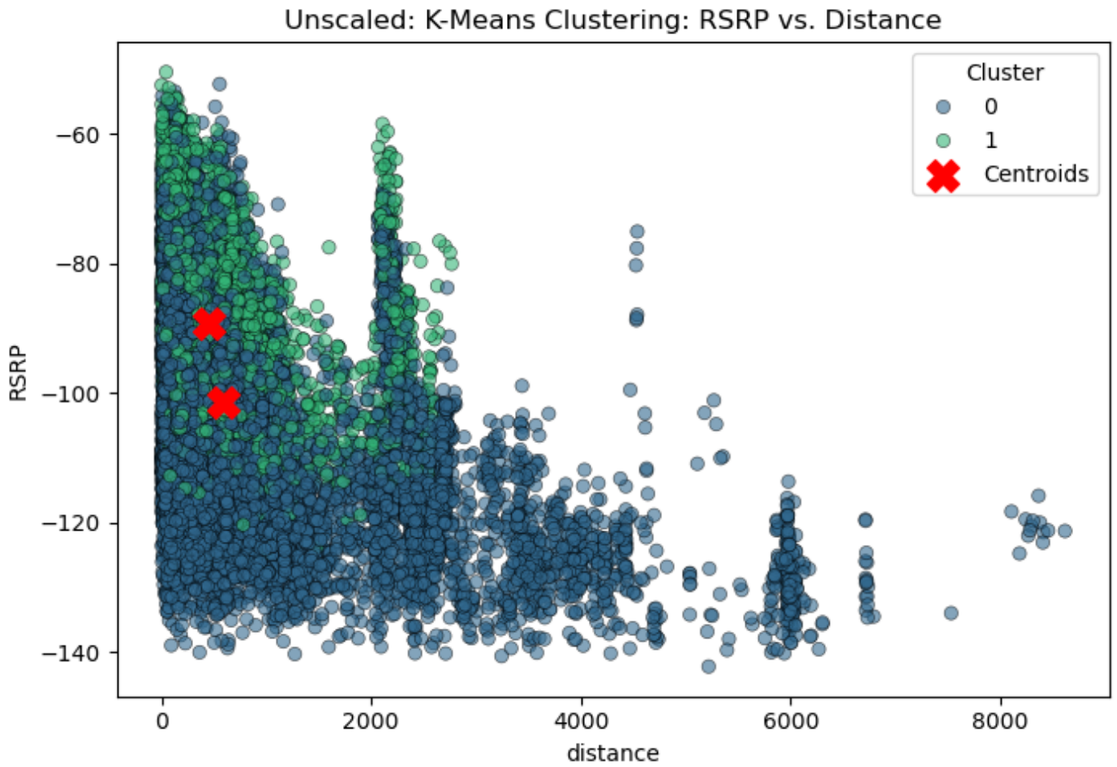

Scaled: K-Means Clustering: RSRP vs. Speed
RSRP
Speed
Cluster
0
1
2
3
Centroids
3
2
1
0
−1
−2
−3
0
1
2
3
4
5
6
7

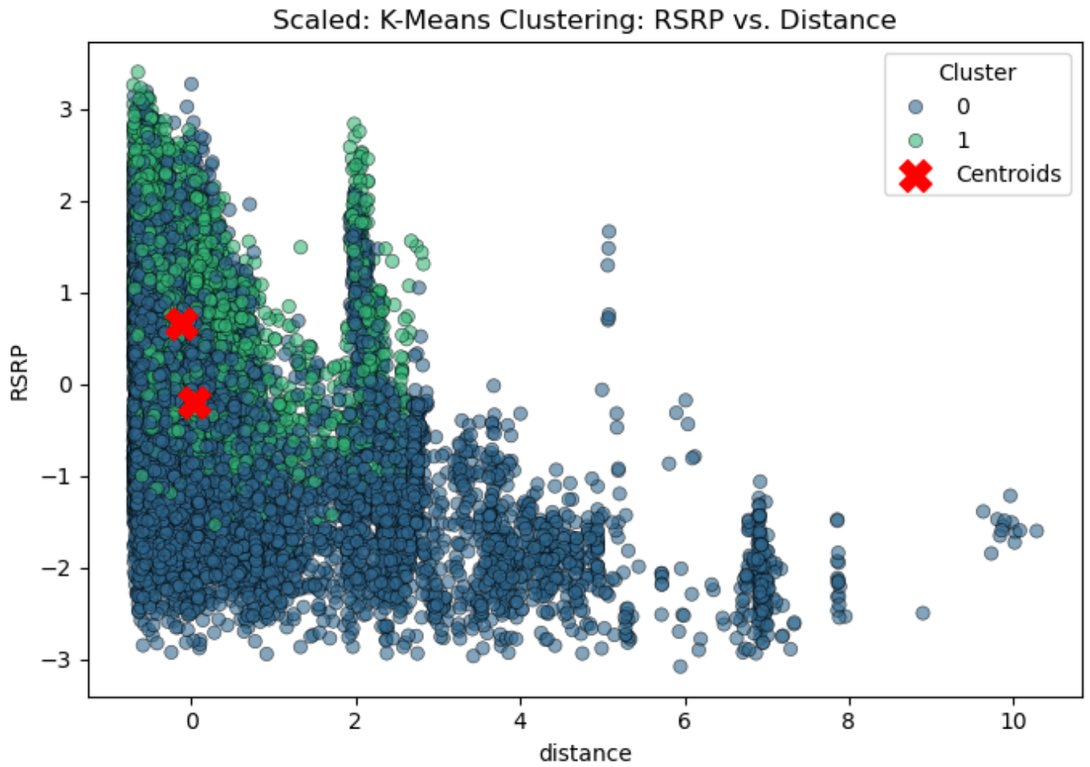


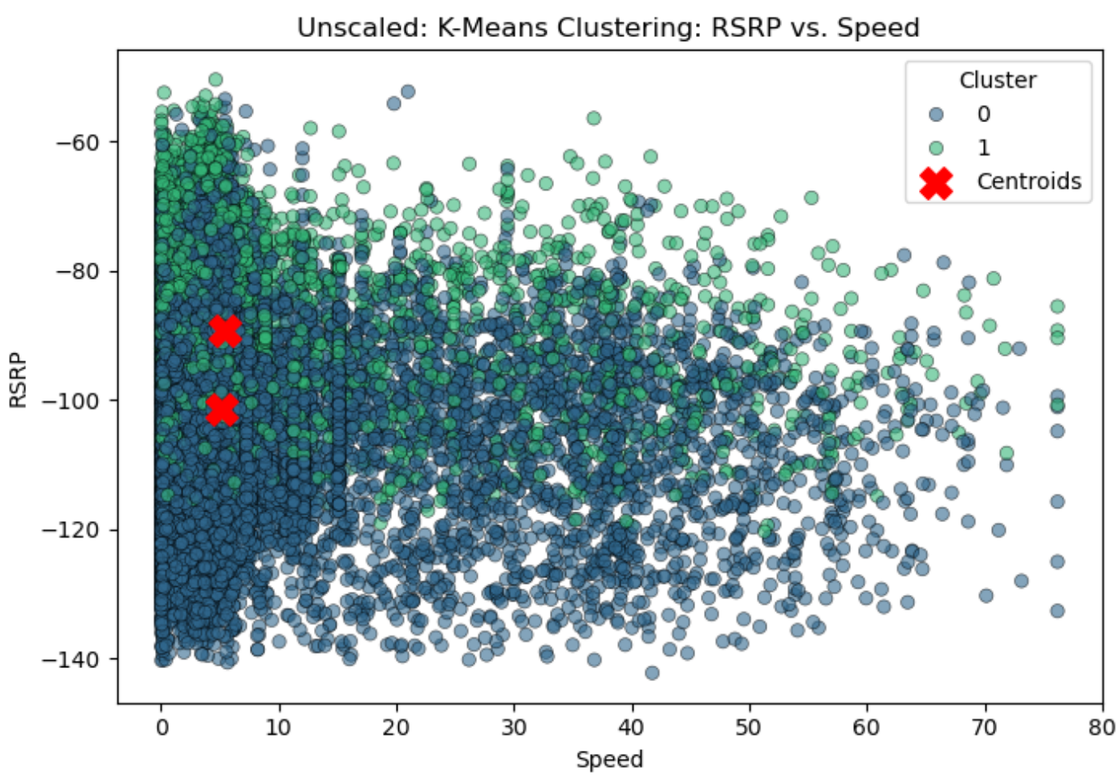


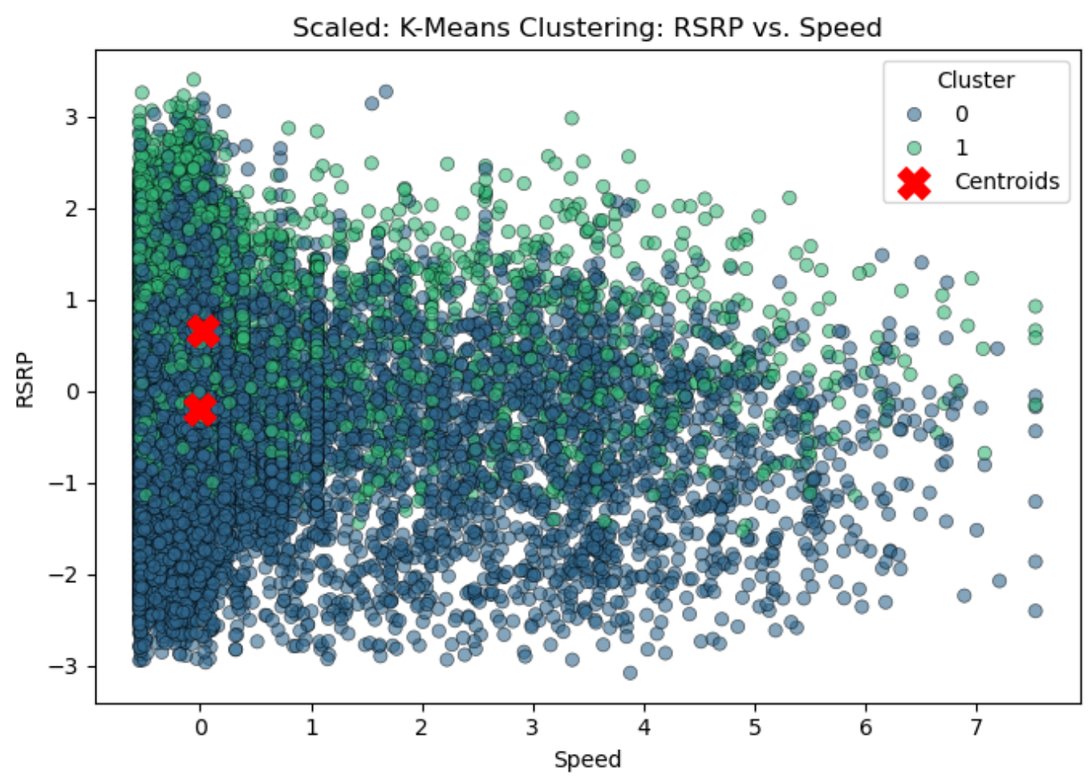
In [66]:
```python
# Select only numeric features
df_numeric = df_original.select_dtypes(include=["number"])

# Scale numeric features
scaler = StandardScaler()
df_scaled = pd.DataFrame(scaler.fit_transform(df_numeric), columns=df_numeric.columns)

# Apply KMeans clustering with 2 clusters
kmeans = KMeans(n_clusters=2, random_state=42, n_init=10)
kmeans.fit(df_scaled)

# Step 4: Assign cluster labels
df_scaled["KMeans_Cluster"] = kmeans.labels_
df_original["KMeans_Cluster"] = kmeans.labels_

# Convert centroids back to original scale
centroids_scaled = kmeans.cluster_centers_
centroids_original = scaler.inverse_transform(centroids_scaled)

# Step 6: Use get_loc for safe column indexing
distance_idx = df_numeric.columns.get_loc("distance")
rsrp_idx     = df_numeric.columns.get_loc("RSRP")
speed_idx    = df_numeric.columns.get_loc("Speed")
band_idx     = df_numeric.columns.get_loc("Band")

# Extract centroid coordinates for plotting
centroids_plot_distance = centroids_original[:, [distance_idx, rsrp_idx]]
centroids_scaled_distance = centroids_scaled[:, [distance_idx, rsrp_idx]]
```

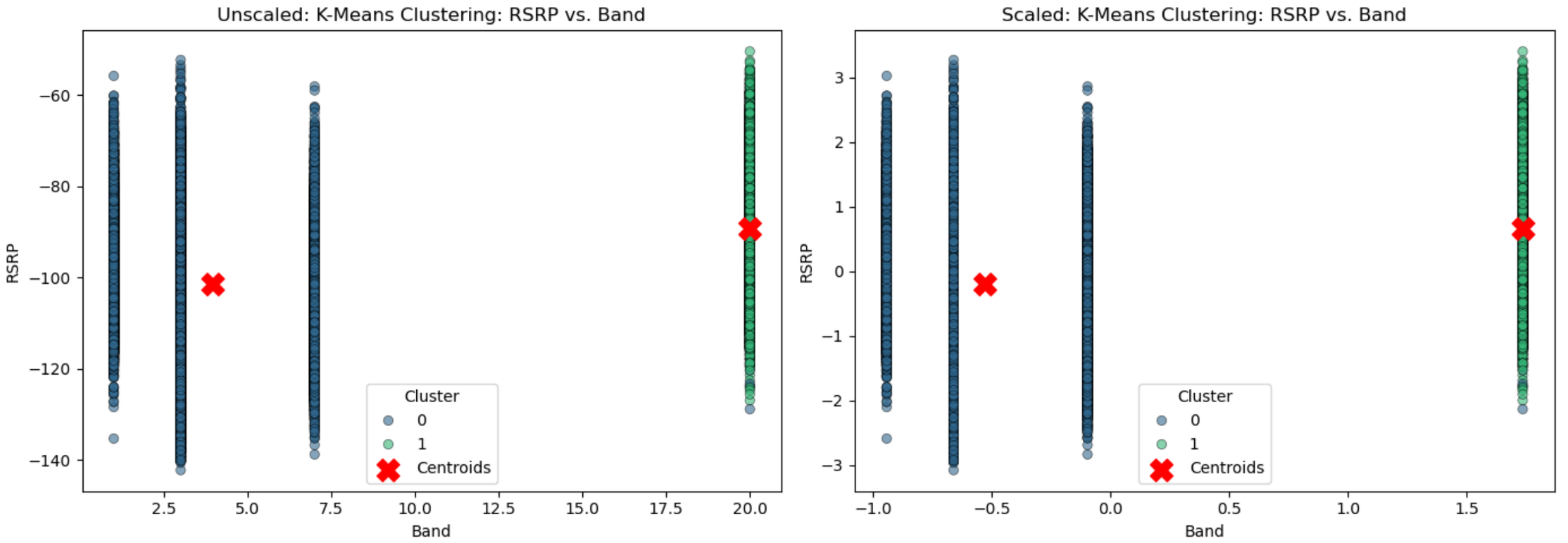
```
centroids_plot_speed = centroids_original[:, [speed_idx, rsrp_idx]]
centroids_scaled_speed = centroids_scaled[:, [speed_idx, rsrp_idx]]

centroids_plot_band = centroids_original[:, [band_idx, rsrp_idx]]
centroids_scaled_band = centroids_scaled[:, [band_idx, rsrp_idx]]

# Define side-by-side plotting function
def plot_clusters_side_by_side(df_original, df_scaled, x_feature, y_feature, cluster_column, title, centroids_original=None, centroids_scaled=None):
    fig, axes = plt.subplots(1, 2, figsize=(14, 5))

    sns.scatterplot(
        data=df_original, x=x_feature, y=y_feature, hue=cluster_column,
        palette='viridis', alpha=0.6, edgecolor='k', ax=axes[0]
    )
    if centroids_original is not None:
        axes[0].scatter(centroids_original[:, 0], centroids_original[:, 1], marker='X', s=200, color='red', label='Centroids')
    axes[0].set_title(f"Unscaled: {title}")
    axes[0].legend(title='Cluster')
    axes[0].set_xlabel(x_feature)
    axes[0].set_ylabel(y_feature)

    sns.scatterplot(
        data=df_scaled, x=x_feature, y=y_feature, hue=cluster_column,
        palette='viridis', alpha=0.6, edgecolor='k', ax=axes[1]
    )
    if centroids_scaled is not None:
        axes[1].scatter(centroids_scaled[:, 0], centroids_scaled[:, 1], marker='X', s=200, color='red', label='Centroids')
    axes[1].set_title(f"Scaled: {title}")
    axes[1].legend(title='Cluster')
    axes[1].set_xlabel(x_feature)
    axes[1].set_ylabel(y_feature)

    plt.tight_layout()
    plt.show()

# Plot different feature combinations
plot_clusters_side_by_side(df_original, df_scaled, 'distance', 'RSRP', 'KMeans_Cluster',
                           'K-Means Clustering: RSRP vs. Distance', centroids_plot_distance, centroids_scaled_distance)

plot_clusters_side_by_side(df_original, df_scaled, 'Speed', 'RSRP', 'KMeans_Cluster',
                           'K-Means Clustering: RSRP vs. Speed', centroids_plot_speed, centroids_scaled_speed)

plot_clusters_side_by_side(df_original, df_scaled, 'Band', 'RSRP', 'KMeans_Cluster',
                           'K-Means Clustering: RSRP vs. Band', centroids_plot_band, centroids_scaled_band)
```


Unscaled: K-Means Clustering: RSRP vs. Distance
Cluster
0
1
Centroids
RSRP
−60
−80
−100
−120
−140
0
2000
4000
6000
8000
distance


Scaled: K-Means Clustering: RSRP vs. Distance
Cluster
0
1
Centroids
RSRP
3
2
1
0
−1
−2
−3
0
2
4
6
8
10
distance

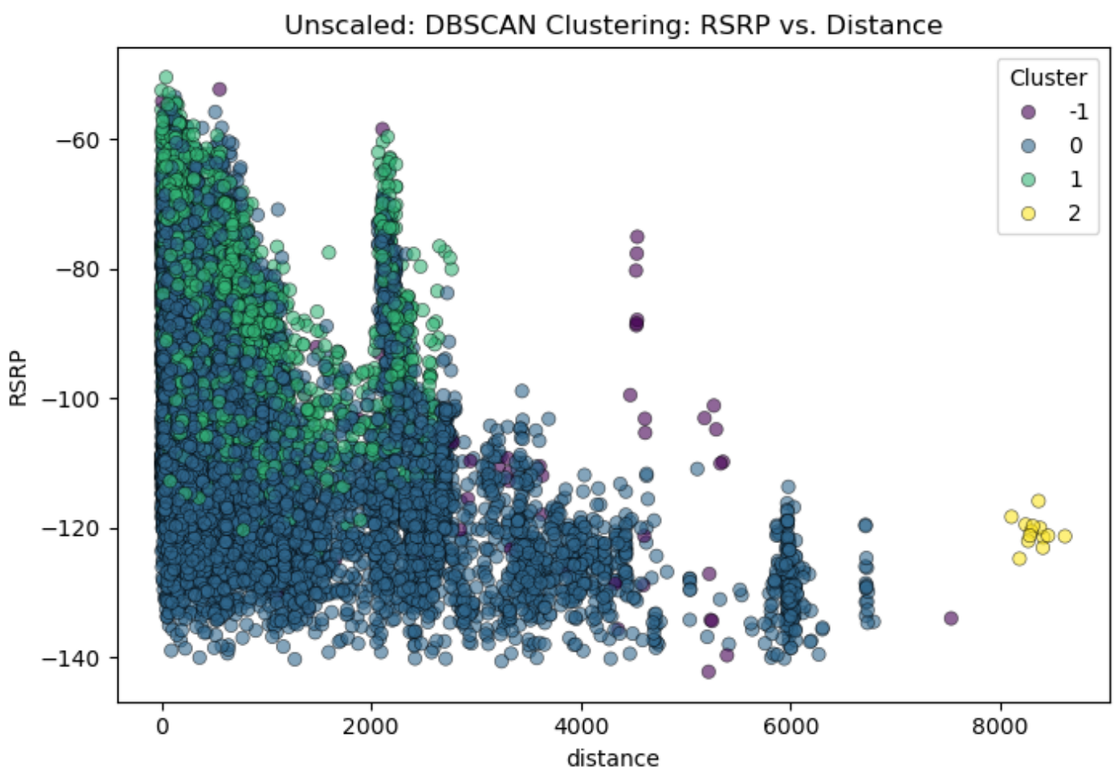

Unscaled: K-Means Clustering: RSRP vs. Speed
Cluster
0
1
Centroids
RSRP
−60
−80
−100
−120
−140
0
10
20
30
40
50
60
70
80
Speed

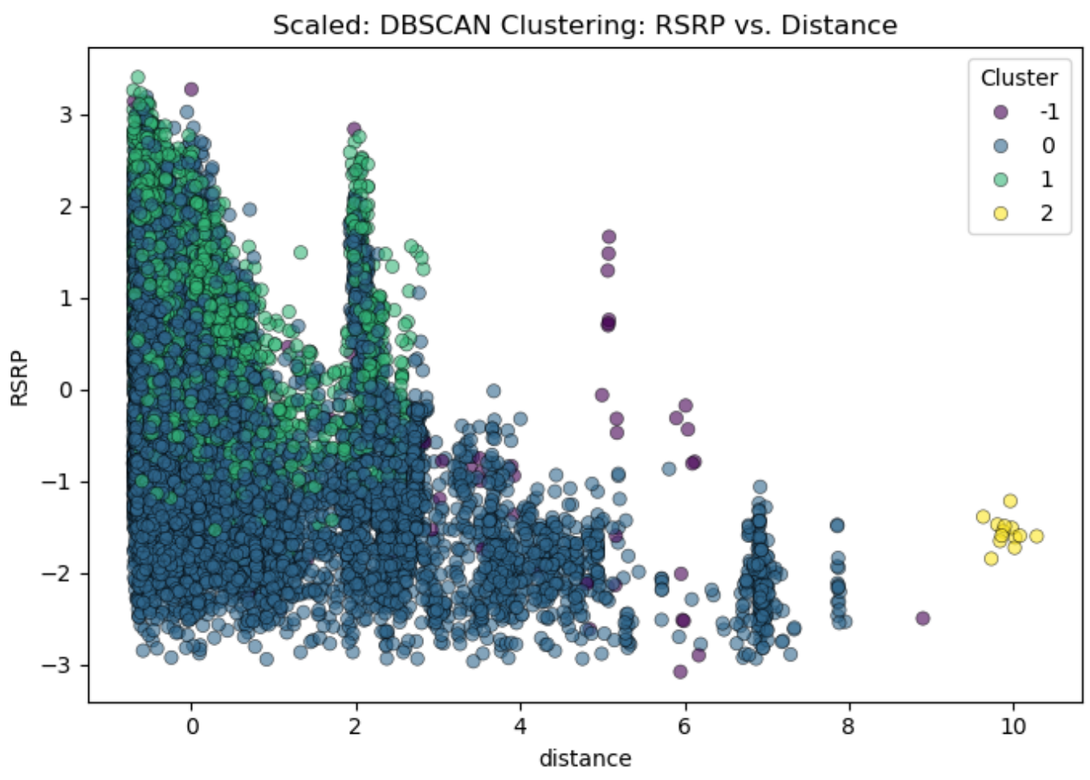

Scaled: K-Means Clustering: RSRP vs. Speed
Cluster
0
1
Centroids
RSRP
3
2
1
0
−1
−2
−3
0
1
2
3
4
5
6
7
Speed

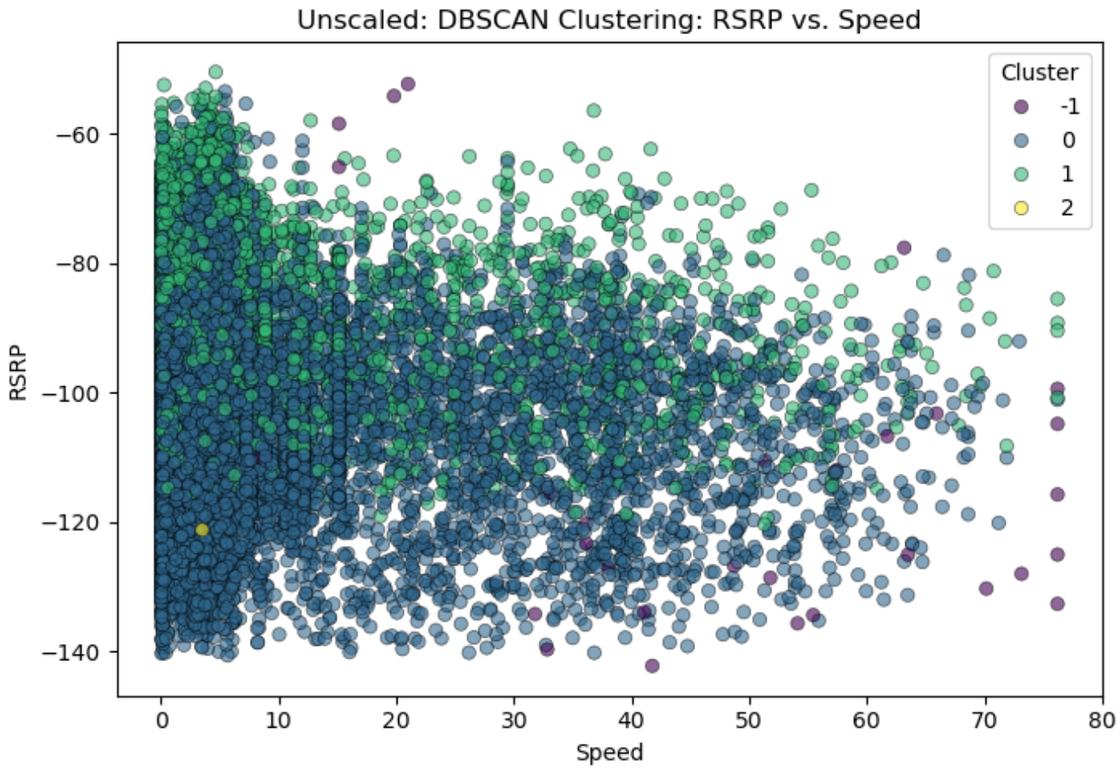


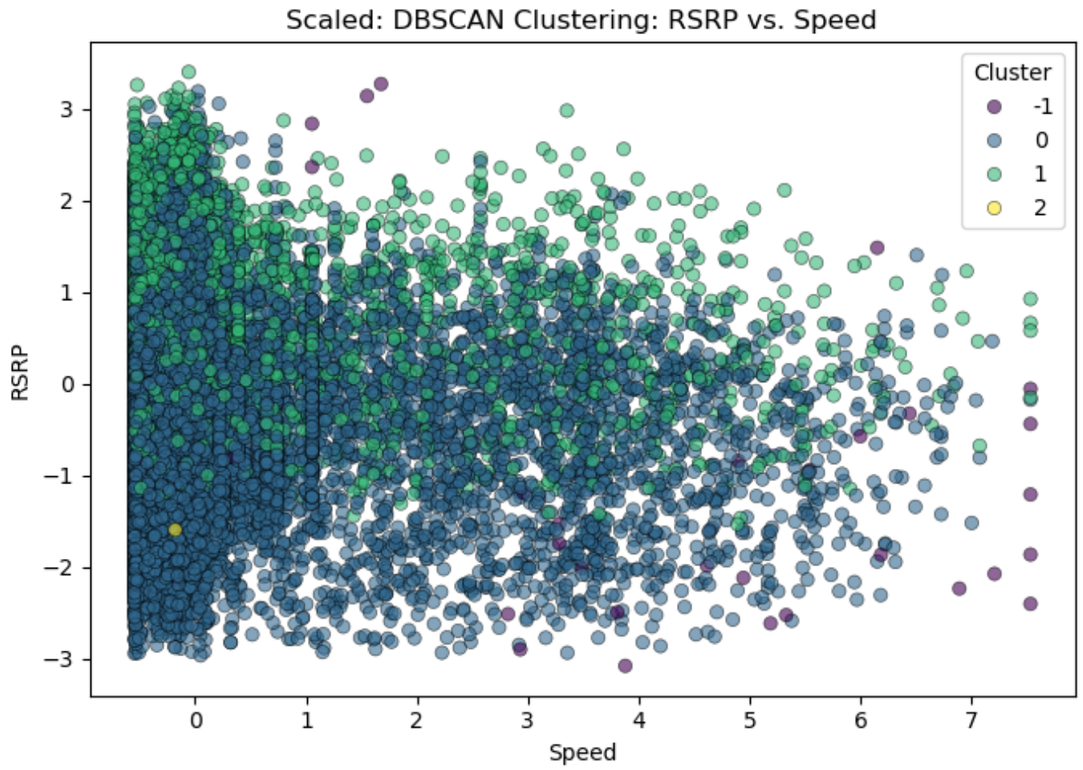
In [67]:
```python
# Assuming df_clustered contains both numerical and categorical features
df_original = df_clustered.copy()

# Select numerical features for scaling
numeric_features = ['distance', 'RSRP', 'Band', 'Speed']  # Add other relevant features
scaler = StandardScaler()

# Scale only numerical features and convert back to DataFrame
scaled_array = scaler.fit_transform(df_original[numeric_features])
df_scaled = pd.DataFrame(scaled_array, columns=numeric_features, index=df_original.index)

# Ensure the cluster labels are retained in the scaled dataset
df_scaled["DBSCAN_Cluster"] = df_original["DBSCAN_Cluster"]

def plot_clusters_side_by_side(df_original, df_scaled, x_feature, y_feature, cluster_column, title):
    """
    Plots scaled and unscaled scatter plots side by side for better visualization.

    Parameters:
        df_original (DataFrame): The original (unscaled) dataset.
        df_scaled (DataFrame): The scaled dataset.
        x_feature (str): Feature for x-axis.
        y_feature (str): Feature for y-axis.
        cluster_column (str): Column representing clusters.
        title (str): Title for the plots.
    """
    fig, axes = plt.subplots(1, 2, figsize=(14, 5))  # Side-by-side plots
```

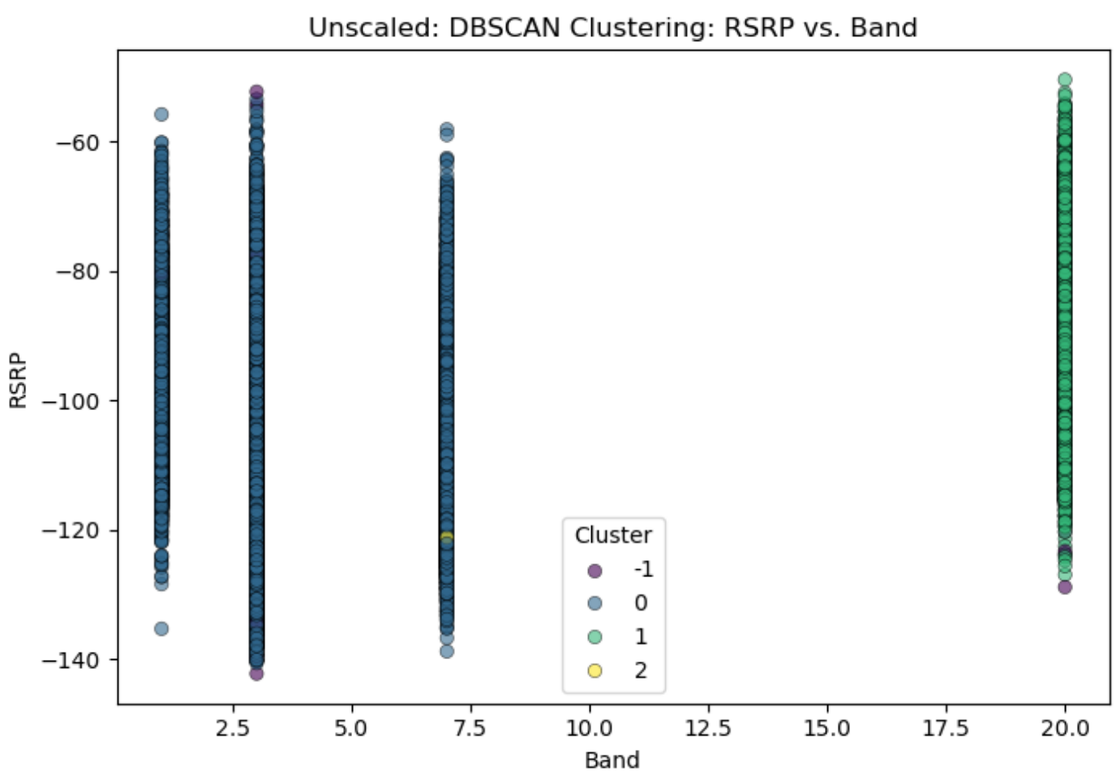
```
    # Unscaled Scatter Plot
    sns.scatterplot(
        data=df_original, x=x_feature, y=y_feature, hue=cluster_column, palette='viridis', alpha=0.6, edgecolor='k', ax=axes[0]
    )
    axes[0].set_title(f"Unscaled: {title}")
    axes[0].set_xlabel(x_feature)
    axes[0].set_ylabel(y_feature)
    axes[0].legend(title='Cluster')

    # Scaled Scatter Plot
    sns.scatterplot(
        data=df_scaled, x=x_feature, y=y_feature, hue=cluster_column, palette='viridis', alpha=0.6, edgecolor='k', ax=axes[1]
    )
    axes[1].set_title(f"Scaled: {title}")
    axes[1].set_xlabel(x_feature)
    axes[1].set_ylabel(y_feature)
    axes[1].legend(title='Cluster')

    plt.tight_layout()
    plt.show()

# Example usage: Compare scaled vs. unscaled clustering results
plot_clusters_side_by_side(df_original, df_scaled, 'distance', 'RSRP', 'DBSCAN_Cluster', 'DBSCAN Clustering: RSRP vs. Distance')
plot_clusters_side_by_side(df_original, df_scaled, 'Speed', 'RSRP', 'DBSCAN_Cluster', 'DBSCAN Clustering: RSRP vs. Speed')
plot_clusters_side_by_side(df_original, df_scaled, 'Band', 'RSRP', 'DBSCAN_Cluster', 'DBSCAN Clustering: RSRP vs. Band')
```

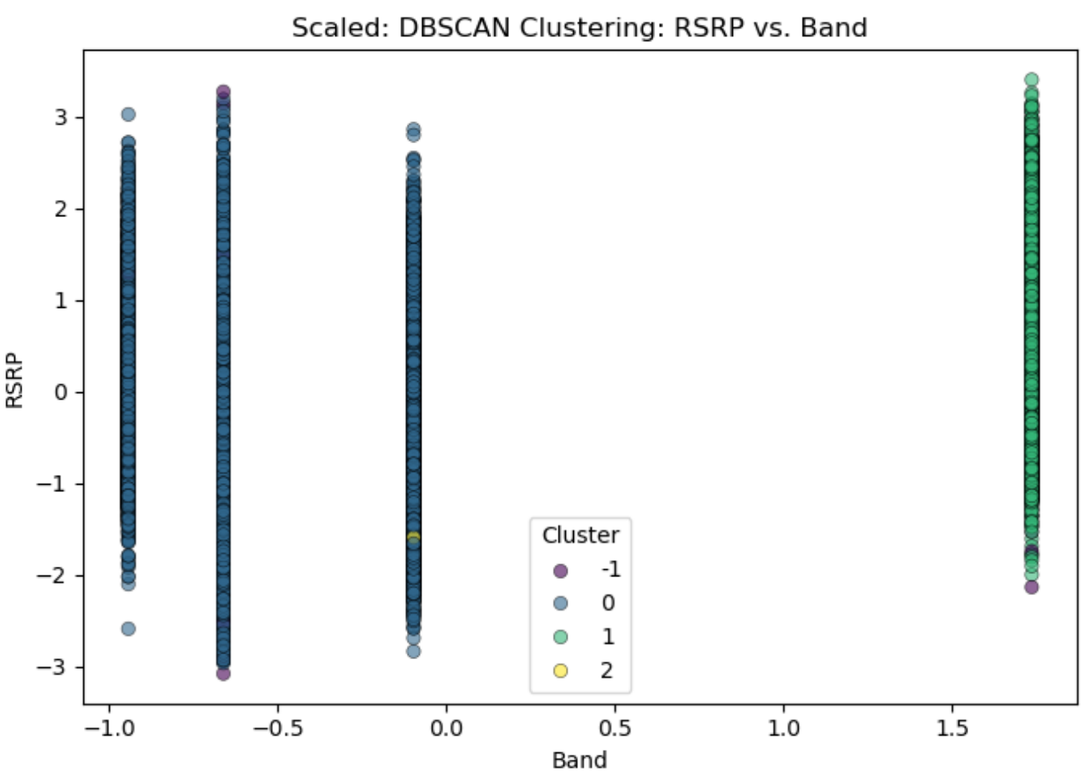





Unscaled: DBSCAN Clustering: RSRP vs. Speed
RSRP
Speed
Cluster
-1
0
1
2


Scaled: DBSCAN Clustering: RSRP vs. Speed
RSRP
Speed
Cluster
-1
0
1
2


Unscaled: DBSCAN Clustering: RSRP vs. Band
RSRP
Band
Cluster
-1
0
1
2


Scaled: DBSCAN Clustering: RSRP vs. Band
RSRP
Band
Cluster
-1
0
1
2

TASK 3

CLASSIFICATION

In [69]:
```
df_classification = df_sample.copy()
```

In [70]:
```
# Verify the sample size
print("Sample shape:", df_classification.shape)
df_classification.info()
```

```
Sample shape: (40000, 26)
<class 'pandas.core.frame.DataFrame'>
Index: 40000 entries, 429425 to 137755
Data columns (total 26 columns):
 #   Column               Non-Null Count  Dtype  
---  ------               --------------  -----  
 0   Date                 40000 non-null  object 
 1   Time                 40000 non-null  object 
 2   UTC                  40000 non-null  float64
 3   Latitude             40000 non-null  float64
 4   Longitude            40000 non-null  float64
 5   Altitude             40000 non-null  float64
 6   Speed                40000 non-null  float64
 7   EARFCN               40000 non-null  int64  
 8   Frequency            40000 non-null  float64
 9   PCI                  40000 non-null  int64  
 10  MNC                  40000 non-null  object 
 11  CellIdentity         40000 non-null  int64  
 12  eNodeB.ID            40000 non-null  int64  
 13  Power                40000 non-null  float64
 14  SINR                 40000 non-null  float64
 15  RSRP                 40000 non-null  float64
 16  RSRQ                 40000 non-null  float64
 17  scenario             40000 non-null  object 
 18  cellLongitude        40000 non-null  float64
 19  cellLatitude         40000 non-null  float64
 20  cellPosErrorLambda1  40000 non-null  float64
 21  cellPosErrorLambda2  40000 non-null  float64
 22  n_CellIdentities     40000 non-null  int64  
 23  distance             40000 non-null  float64
 24  Band                 40000 non-null  int64  
 25  campaign             40000 non-null  object 
dtypes: float64(15), int64(6), object(5)
memory usage: 8.2+ MB
```

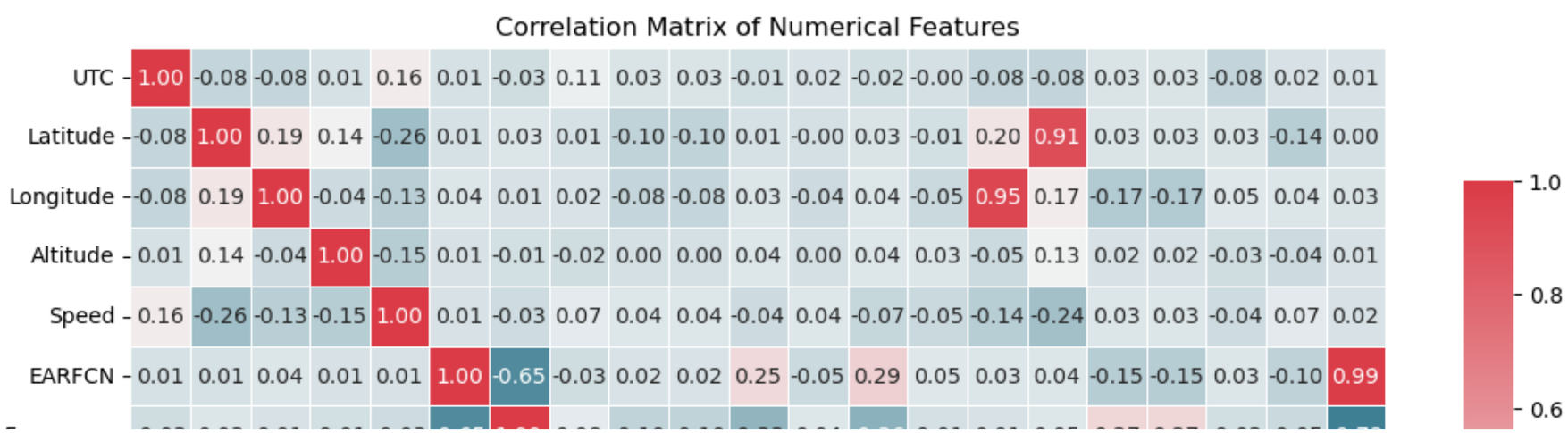
In [71]:
```
# Check unique values and their counts in the 'MNC', 'scenario', and 'campaign' columns before encoding
unique_mnc = df['MNC'].unique()
unique_scenario = df['scenario'].unique()
unique_campaign = df['campaign'].unique()

# Create a data frame to hold the unique values and their counts
data = {
    'Feature': ['MNC', 'scenario', 'campaign'],
```

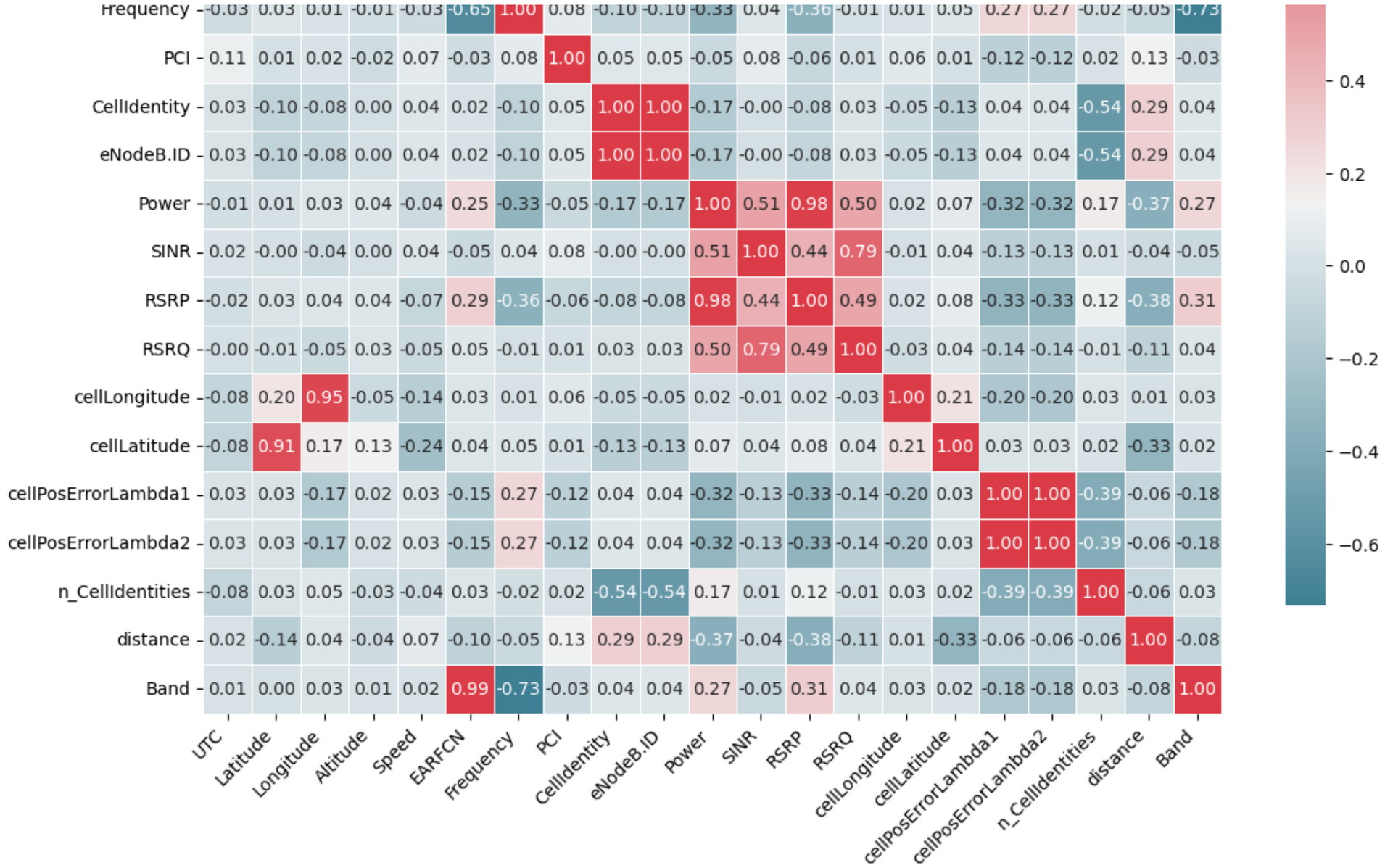
```
    'Unique Values': [unique_mnc, unique_scenario, unique_campaign],
    'Count of Unique Values': [len(unique_mnc), len(unique_scenario), len(unique_campaign)],
}

df_unique = pd.DataFrame(data)

# Display the DataFrame
print(df_unique)
```

```
    Feature                                      Unique Values  \
0       MNC                                  ["Op"[1], "Op"[2]]
1  scenario                                        [OW, IS, OD]
2  campaign  [campaign_6_OW_4G_gaming, campaign_35_IS_4G_sp...

   Count of Unique Values
0                       2
1                       3
2                     193
```

In [72]:

```
# Feature Selection
# Select only numeric columns
numeric_df = df_classification.select_dtypes(include=['number'])
corrmat = numeric_df.corr()
# Set up the matplotlib figure
fig, ax = plt.subplots(figsize=(12, 10))  # Adjust the size as needed
# Generate a custom diverging colormap
cmap = sns.diverging_palette(220, 10, as_cmap=True)
# Draw the heatmap with the full correlation matrix
sns.heatmap(corrmat, cmap=cmap, annot=True, fmt='.2f', linewidths=0.5,
            cbar_kws={"shrink": 0.75}, annot_kws={"size": 10},  # Adjust annotation font size
            square=True, ax=ax)
# Rotate the tick labels for better readability
plt.xticks(rotation=45, ha='right', fontsize=10)
plt.yticks(rotation=0, fontsize=10)
# Set the title with appropriate font size
plt.title('Correlation Matrix of Numerical Features', fontsize=12)
# Adjust layout to ensure everything fits without overlap
plt.tight_layout()
# Display the heatmap
plt.show()
```





In [73]:
```python
# Select numeric columns for ANOVA
numeric_columns = numeric_df.columns

# Ensure categorical variables exist
if 'MNC' in df_classification.columns and 'scenario' in df_classification.columns:
    # Perform ANOVA for 'MNC' against numeric features
    anova_results_mnc = {col: stats.f_oneway(*[numeric_df[col][df_classification['MNC'] == mnc]
                                               for mnc in df_classification['MNC'].unique()])
                         for col in numeric_columns}

    # Perform ANOVA for 'scenario' against numeric features
    anova_results_scenario = {col: stats.f_oneway(*[numeric_df[col][df_classification['scenario'] == scenario]
                                                    for scenario in df_classification['scenario'].unique()])
                              for col in numeric_columns}
```

```
    # Convert results to a DataFrame
    anova_mnc_df = pd.DataFrame([(col, result.statistic, result.pvalue)
                                 for col, result in anova_results_mnc.items()],
                                columns=['Feature', 'F-Statistic (MNC)', 'p-Value (MNC)'])

    anova_scenario_df = pd.DataFrame([(col, result.statistic, result.pvalue)
                                      for col, result in anova_results_scenario.items()],
                                     columns=['Feature', 'F-Statistic (Scenario)', 'p-Value (Scenario)'])

    # Merge results
    anova_results_df = pd.merge(anova_mnc_df, anova_scenario_df, on='Feature')

    # Display ANOVA results
    print(anova_results_df)
else:
    print("Columns 'MNC' or 'scenario' not found in df_classification.")
```

	Feature	F-Statistic (MNC)	p-Value (MNC)	F-Statistic (Scenario)	p-Value (Scenario)
0	UTC	2.819583e+01	1.102247e-07	2731.681281	0.000000e+00
1	Latitude	3.600559e+02	6.154144e-80	2016.911303	0.000000e+00
2	Longitude	1.783268e+02	1.373380e-40	2857.889487	0.000000e+00
3	Altitude	7.814768e-03	9.295583e-01	660.675088	5.129718e-283
4	Speed	6.758405e+01	2.079092e-16	9551.908112	0.000000e+00
5	EARFCN	4.081763e+01	1.689523e-10	13.455401	1.439987e-06
6	Frequency	4.337494e+02	8.020583e-96	78.723196	7.552070e-35
7	PCI	9.899816e+01	2.690060e-23	1267.503117	0.000000e+00
8	CellIdentity	2.615513e+06	0.000000e+00	158.286867	3.369061e-69
9	eNodeB.ID	2.615517e+06	0.000000e+00	158.286951	3.368778e-69
10	Power	1.083579e+03	1.676666e-234	862.203189	0.000000e+00
11	SINR	7.175527e-01	3.969529e-01	27.355836	1.341621e-12
12	RSRP	2.218956e+02	4.756339e-50	891.972287	0.000000e+00
13	RSRQ	1.992132e+01	8.091562e-06	186.379364	2.699772e-81
14	cellLongitude	5.034379e+01	1.311781e-12	2596.132701	0.000000e+00
15	cellLatitude	6.776062e+02	3.815888e-148	2217.686672	0.000000e+00
16	cellPosErrorLambda1	6.505639e+01	7.479449e-16	337.205535	5.949057e-146
17	cellPosErrorLambda2	6.355954e+01	1.596925e-15	336.756245	9.254198e-146
18	n_CellIdentities	1.758485e+04	0.000000e+00	132.159941	6.201362e-58
19	distance	3.585214e+03	0.000000e+00	451.696453	1.031121e-194
20	Band	1.011488e+02	9.106608e-24	18.129791	1.348648e-08

In [74]:
```
# Create a copy of the original dataset
df1_encoded = df_classification.copy()

# One-Hot Encoding for 'MNC' and 'scenario'
df1_encoded = pd.get_dummies(df1_encoded, columns=['MNC', 'scenario'], drop_first=True)

# Convert specific one-hot encoded columns to integers
for col in df1_encoded.columns:
    if col.startswith('MNC_') or col.startswith('scenario_'):
        df1_encoded[col] = df1_encoded[col].astype(int)
```

```
# Encoding 'campaign'
campaign_unique = df1_encoded['campaign'].nunique()
print("Number of unique campaigns:", campaign_unique)

if campaign_unique < 10:
    df1_encoded = pd.get_dummies(df1_encoded, columns=['campaign'], drop_first=True)
else:
    le_campaign = LabelEncoder()
    df1_encoded['campaign_encoded'] = le_campaign.fit_transform(df1_encoded['campaign'])
    df1_encoded.drop(columns=['campaign'], inplace=True)

# Process Date and Time Columns
df1_encoded['Date'] = pd.to_datetime(df1_encoded['Date'], format='%d.%m.%Y', errors='coerce')

df1_encoded['Time'] = pd.to_datetime(df1_encoded['Time'], format='%H:%M:%S.%f', errors='coerce')
df1_encoded['Time'] = df1_encoded['Time'].dt.time  # Extract only the time part

# Check the resulting DataFrame
print(df1_encoded.head())
```

```
Number of unique campaigns: 193
            Date             Time           UTC   Latitude  Longitude  \
429425 2021-01-08  17:01:28.658000  1.612800e+09  41.875349  12.522386
371599 2020-12-17  18:31:31.251000  1.610905e+09  41.889238  12.493533
125590 2021-01-06  10:24:25.860000  1.612603e+09  41.872970  12.525257
508237 2020-12-22  13:32:22.699000  1.610579e+09  41.893862  12.494724
64415  2021-01-11  17:26:35.887000  1.613061e+09  41.823753  12.467389

          Altitude  Speed  EARFCN  Frequency  PCI  ...  cellLatitude  \
429425   69.770000  35.68    1350     1820.0  426  ...     41.876179
371599   39.460000   0.07    3025     2647.5  427  ...     41.889008
125590  116.260000   2.34    6300      806.0  351  ...     41.882354
508237   69.665625   0.61    6400      816.0  178  ...     41.895079
64415    42.090000   0.04    6400      816.0  111  ...     41.830381

        cellPosErrorLambda1  cellPosErrorLambda2  n_CellIdentities  \
429425             0.358485             0.249708                 5
371599             0.322294             0.194148                10
125590             0.290521             0.188450                 6
508237             5.990001             5.990001                12
64415              0.213255             0.167991                13

           distance  Band  MNC_"Op"[2]  scenario_OD  scenario_OW  \
429425   173.311288     3            0            1            0
371599  1006.125012     7            1            1            0
125590  1045.267221    20            0            0            1
508237   158.577997    20            1            0            0
64415    738.752534    20            1            1            0

        campaign_encoded
429425               156
371599               128
125590               110
508237                 4
64415                 27

[5 rows x 27 columns]
```

Binary

In [76]:
```
# Create a copy of the df1 dataset
binary_classification = df1_encoded.copy()
```

In [77]:
```
# Drop the specified columns from the dataframe
columns_to_drop = ["cellLatitude", "cellLongitude", "eNodeB.ID", "CellIdentity", "PCI",
                   "Altitude", "Longitude", "Latitude", "UTC", "Date", "Time", "campaign_encoded"]

# Drop columns if they exist in the dataframe
df_binary_classification = binary_classification.drop(columns=columns_to_drop, errors='ignore')

# Check the resulting DataFrame
df_binary_classification.info()
```

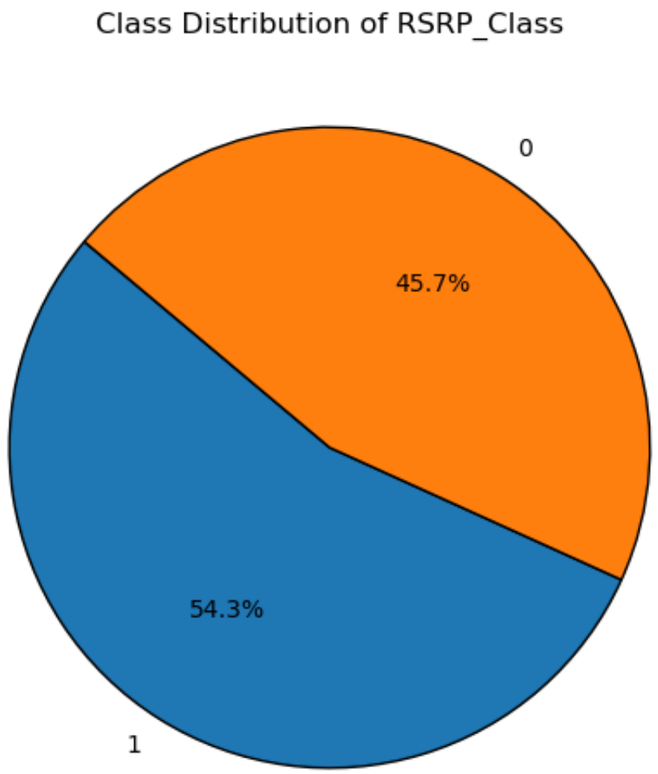
```
<class 'pandas.core.frame.DataFrame'>
Index: 40000 entries, 429425 to 137755
Data columns (total 15 columns):
 #   Column               Non-Null Count  Dtype  
---  ------               --------------  -----  
 0   Speed                40000 non-null  float64
 1   EARFCN               40000 non-null  int64  
 2   Frequency            40000 non-null  float64
 3   Power                40000 non-null  float64
 4   SINR                 40000 non-null  float64
 5   RSRP                 40000 non-null  float64
 6   RSRQ                 40000 non-null  float64
 7   cellPosErrorLambda1  40000 non-null  float64
 8   cellPosErrorLambda2  40000 non-null  float64
 9   n_CellIdentities     40000 non-null  int64  
 10  distance             40000 non-null  float64
 11  Band                 40000 non-null  int64  
 12  MNC_"Op"[2]          40000 non-null  int64  
 13  scenario_OD          40000 non-null  int64  
 14  scenario_OW          40000 non-null  int64  
dtypes: float64(9), int64(6)
memory usage: 4.9 MB
```

In [78]:
```python
# Define the target variable (y)
y = df_binary_classification["RSRP"]

# Define the feature variables (X) by dropping the target column
X = df_binary_classification.drop(columns=["RSRP"])

# Check the shapes of X and y to confirm the split
print("Shape of X:", X.shape)
print("Shape of y:", y.shape)
```

```
Shape of X: (40000, 14)
Shape of y: (40000,)
```

In [79]:
```python
'''
    Define the binary classification based on the RSRP threshold
    1 represents Signal and 0 represents No Signal
'''
df_binary_classification["RSRP_Class"] = df_binary_classification["RSRP"].apply(lambda x: 0 if x <= -100 else 1)

# Verify class distribution
class_counts = df_binary_classification["RSRP_Class"].value_counts()

# Define colors for better visualization
colors = ["#1f77b4", "#ff7f0e"]

# Plot pie chart
plt.figure(figsize=(6, 6))
plt.pie(
    class_counts,
    labels=class_counts.index,
    autopct="%1.1f%%",
    startangle=140,
    colors=colors,
    wedgeprops={"edgecolor": "black"}
```

```
)
plt.title("Class Distribution of RSRP_Class")
plt.show()

# Check the updated DataFrame
print(df_binary_classification[["RSRP", "RSRP_Class"]].head())
```




```
          RSRP  RSRP_Class
429425  -96.38           1
371599 -114.08           0
125590 -103.54           0
508237  -80.70           1
64415  -100.73           0
```

In [80]:
```
# Define the new target variable
y = df_binary_classification["RSRP_Class"]

# Define feature variables (drop original RSRP column)
X = df_binary_classification.drop(columns=["RSRP", "RSRP_Class"])

# Check the shapes
print("Shape of X:", X.shape)
print("Shape of y:", y.shape)
```

```
Shape of X: (40000, 14)
Shape of y: (40000,)
```

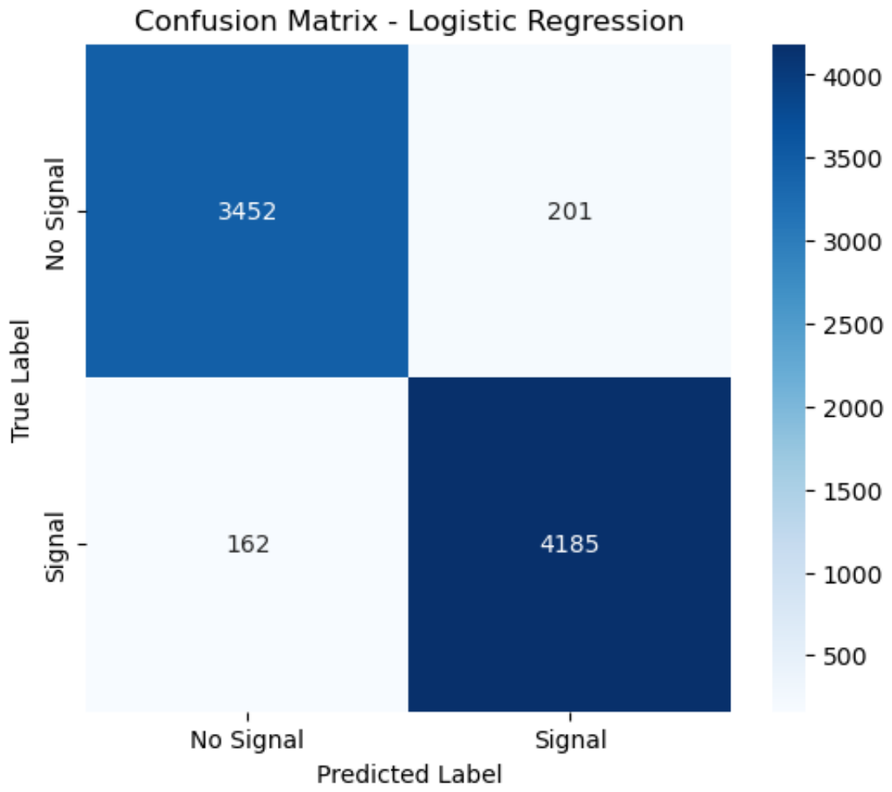
In [81]:
```
# Split data into training and testing sets (80% train, 20% test)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42, stratify=y)

# Standardize the features for both models
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Train Logistic Regression Model (Requires Scaling)
log_reg = LogisticRegression(random_state=42)
log_reg.fit(X_train_scaled, y_train)
y_pred_logreg = log_reg.predict(X_test_scaled)

# Train Random Forest Model (Optional Scaling, but better to keep consistency)
rf_model = RandomForestClassifier(n_estimators=100, random_state=42)
rf_model.fit(X_train_scaled, y_train)
y_pred_rf = rf_model.predict(X_test_scaled)

# Evaluate Models
logreg_acc = accuracy_score(y_test, y_pred_logreg)
rf_acc = accuracy_score(y_test, y_pred_rf)

logreg_report = classification_report(y_test, y_pred_logreg)
rf_report = classification_report(y_test, y_pred_rf)

logreg_conf_matrix = confusion_matrix(y_test, y_pred_logreg)
rf_conf_matrix = confusion_matrix(y_test, y_pred_rf)

# Display results in a DataFrame
results_df = pd.DataFrame({
    "Model": ["Logistic Regression", "Random Forest"],
    "Accuracy": [logreg_acc, rf_acc]
})

# Print the results
print("\nModel Performance Comparison:")
print(results_df.to_string(index=False))
```

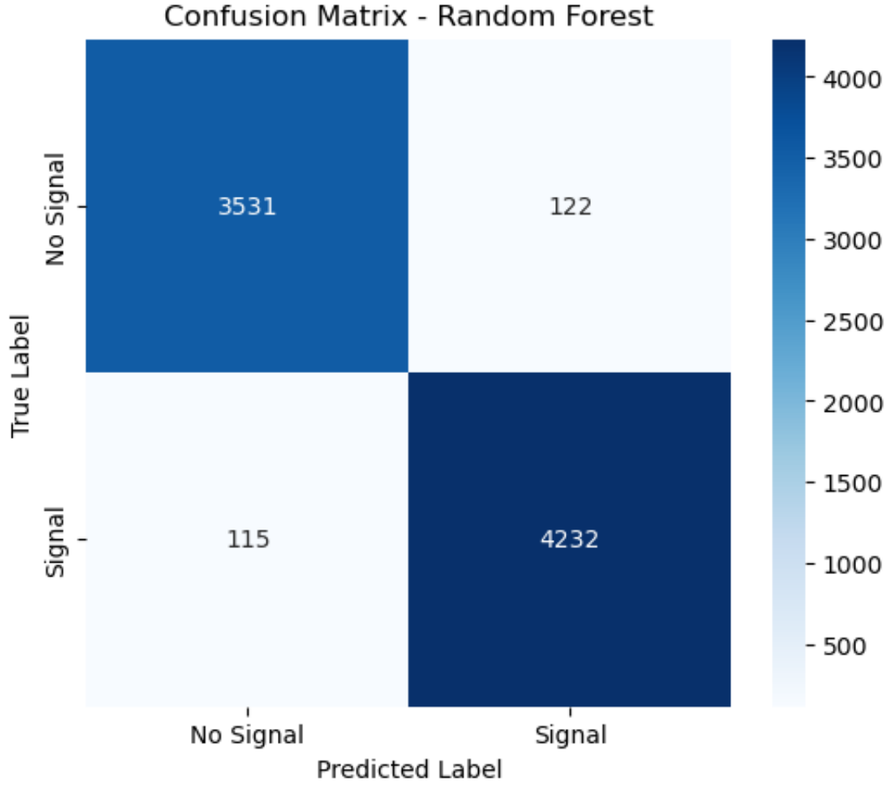
```
Model Performance Comparison:
              Model  Accuracy
Logistic Regression  0.954625
      Random Forest  0.970375
```

In [82]:
```
# Model names and accuracy scores
models = ["Logistic Regression", "Random Forest"]
accuracy_scores = [0.954625, 0.970375]

# Plot model accuracy comparison
plt.figure(figsize=(8, 5))
plt.bar(models, accuracy_scores, color=['red', 'green'])
plt.ylim(0, 1)  # Accuracy ranges from 0 to 1
plt.ylabel("Accuracy Score")
plt.title("Model Performance Comparison")
plt.grid(axis="y", linestyle="--", alpha=0.7)

# Display accuracy values on bars
```

```
for i, acc in enumerate(accuracy_scores):
    plt.text(i, acc + 0.02, f"{acc:.2f}", ha='center', fontsize=12, fontweight='bold')

plt.show()
```

In [83]:
```
# Function to plot confusion matrix
def plot_confusion_matrix(y_true, y_pred, model_name):
    cm = confusion_matrix(y_true, y_pred)
    plt.figure(figsize=(6, 5))
    sns.heatmap(cm, annot=True, fmt="d", cmap="Blues", xticklabels=["No Signal", "Signal"], yticklabels=["No Signal", "Signal"])
    plt.xlabel("Predicted Label")
    plt.ylabel("True Label")
    plt.title(f"Confusion Matrix - {model_name}")
    plt.show()

# Plot Confusion Matrices
plot_confusion_matrix(y_test, y_pred_logreg, "Logistic Regression")
plot_confusion_matrix(y_test, y_pred_rf, "Random Forest")

# Print Classification Reports
print("\n🔷 Logistic Regression Classification Report:\n", classification_report(y_test, y_pred_logreg))
print("\n🔷 Random Forest Classification Report:\n", classification_report(y_test, y_pred_rf))
```

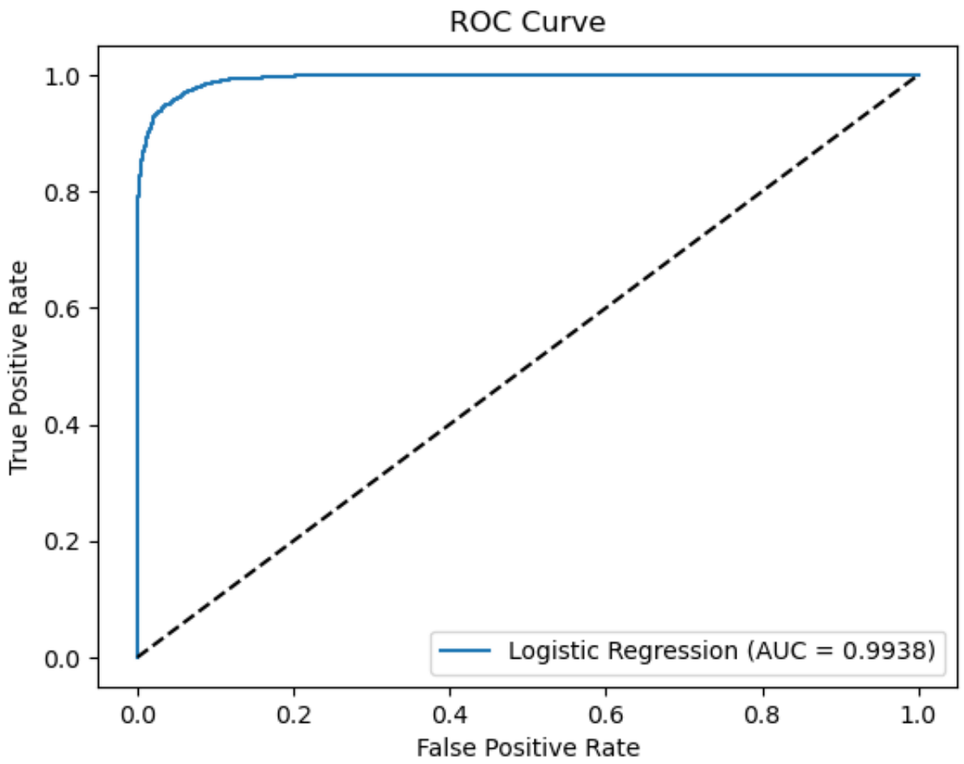

Confusion Matrix - Logistic Regression
No Signal
Signal
True Label
3452
201
162
4185
No Signal
Signal
Predicted Label
4000
3500
3000
2500
2000
1500
1000
500


Confusion Matrix - Random Forest
No Signal
Signal
True Label
3531
122
115
4232
No Signal
Signal
Predicted Label
4000
3500
3000
2500
2000
1500
1000
500

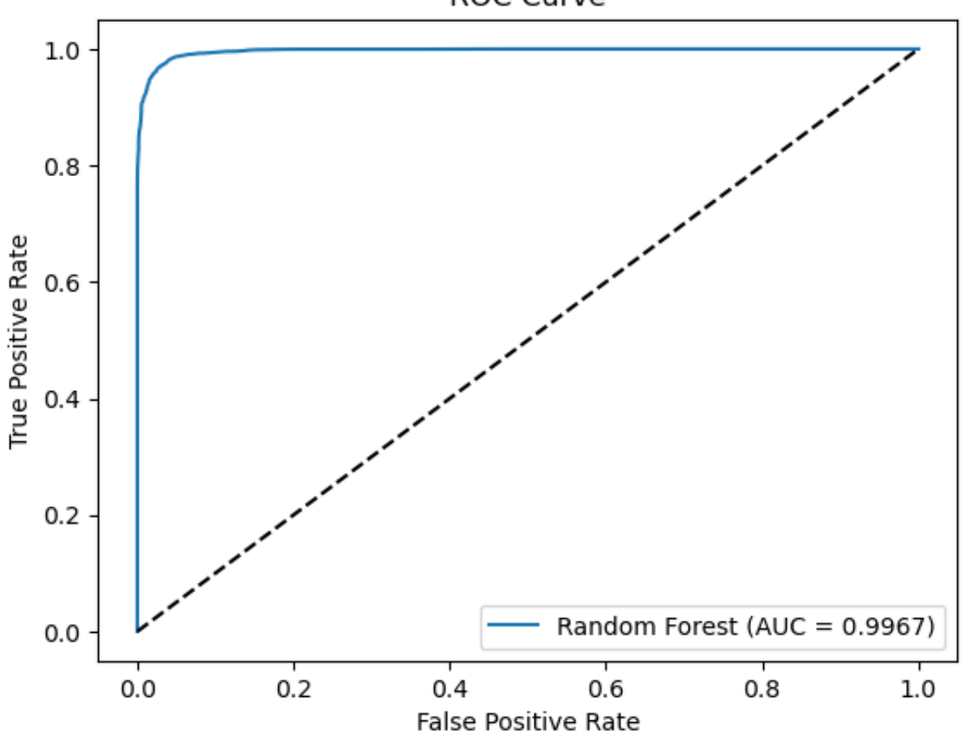
```
🔷 Logistic Regression Classification Report:
              precision    recall  f1-score   support

           0       0.96      0.94      0.95      3653
           1       0.95      0.96      0.96      4347

    accuracy                           0.95      8000
   macro avg       0.95      0.95      0.95      8000
weighted avg       0.95      0.95      0.95      8000


🔷 Random Forest Classification Report:
              precision    recall  f1-score   support

           0       0.97      0.97      0.97      3653
           1       0.97      0.97      0.97      4347

    accuracy                           0.97      8000
   macro avg       0.97      0.97      0.97      8000
weighted avg       0.97      0.97      0.97      8000
```

In [84]:
```python
# Compute probabilities for ROC and Precision-Recall Curves
y_prob_logreg = log_reg.predict_proba(X_test_scaled)[:, 1]
y_prob_rf = rf_model.predict_proba(X_test_scaled)[:, 1]

# Compute ROC-AUC Scores
roc_auc_logreg = roc_auc_score(y_test, y_prob_logreg)
roc_auc_rf = roc_auc_score(y_test, y_prob_rf)

print(f"\nROC-AUC Score (Logistic Regression): {roc_auc_logreg:.4f}")
print(f"ROC-AUC Score (Random Forest): {roc_auc_rf:.4f}")

# Function to plot ROC curves
def plot_roc_curve(y_test, y_prob, model_name):
    fpr, tpr, _ = roc_curve(y_test, y_prob)
    plt.plot(fpr, tpr, label=f"{model_name} (AUC = {auc(fpr, tpr):.4f})")
    plt.plot([0, 1], [0, 1], 'k--')  # Diagonal line (random guessing)
    plt.xlabel("False Positive Rate")
    plt.ylabel("True Positive Rate")
    plt.title("ROC Curve")
    plt.legend()
    plt.show()

# Plot ROC Curves
plot_roc_curve(y_test, y_prob_logreg, "Logistic Regression")
plot_roc_curve(y_test, y_prob_rf, "Random Forest")
```

```
ROC-AUC Score (Logistic Regression): 0.9938
ROC-AUC Score (Random Forest): 0.9967
```



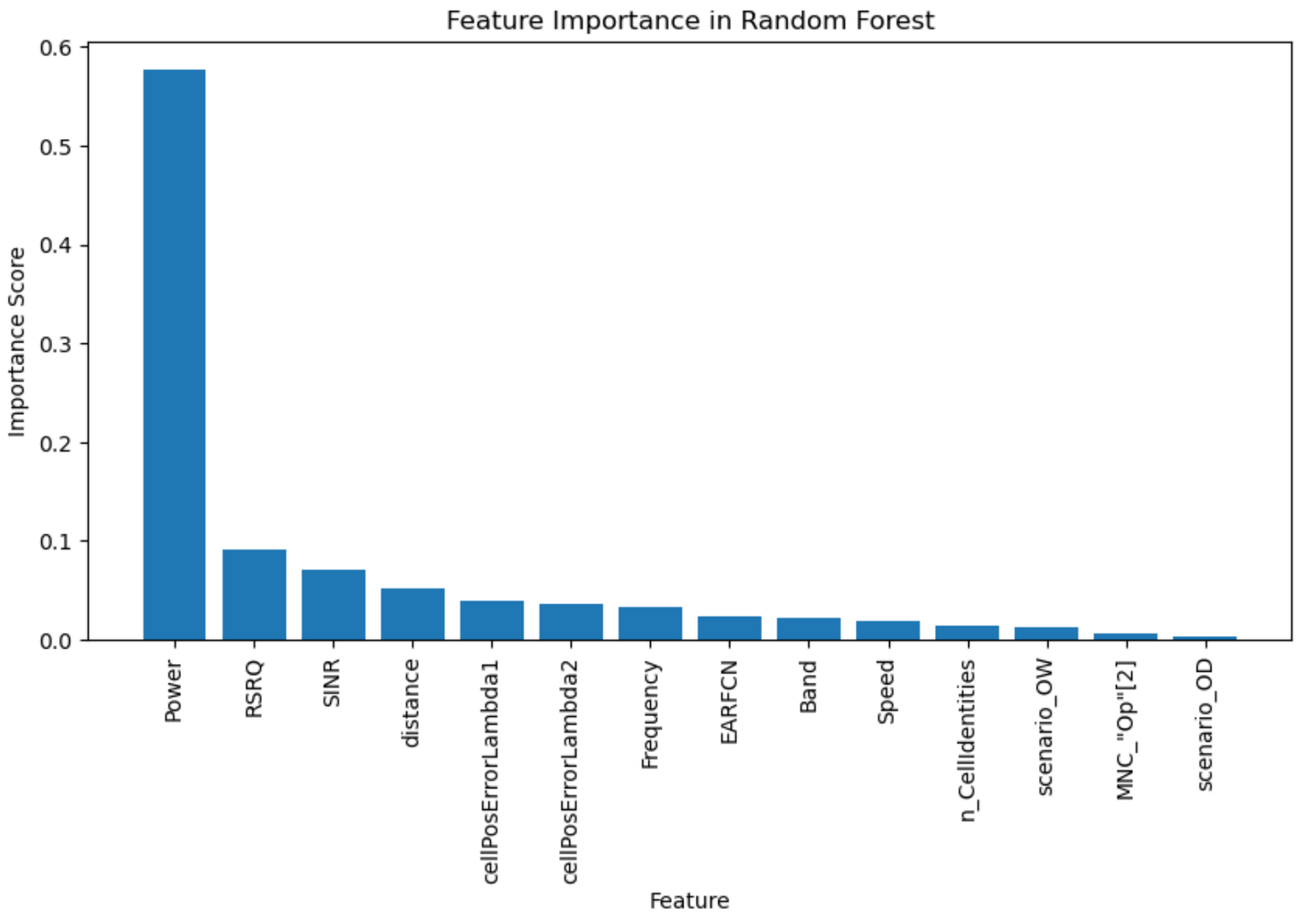
ROC Curve



In [85]:
```
print(X_train_scaled.shape, X_test_scaled.shape)
```

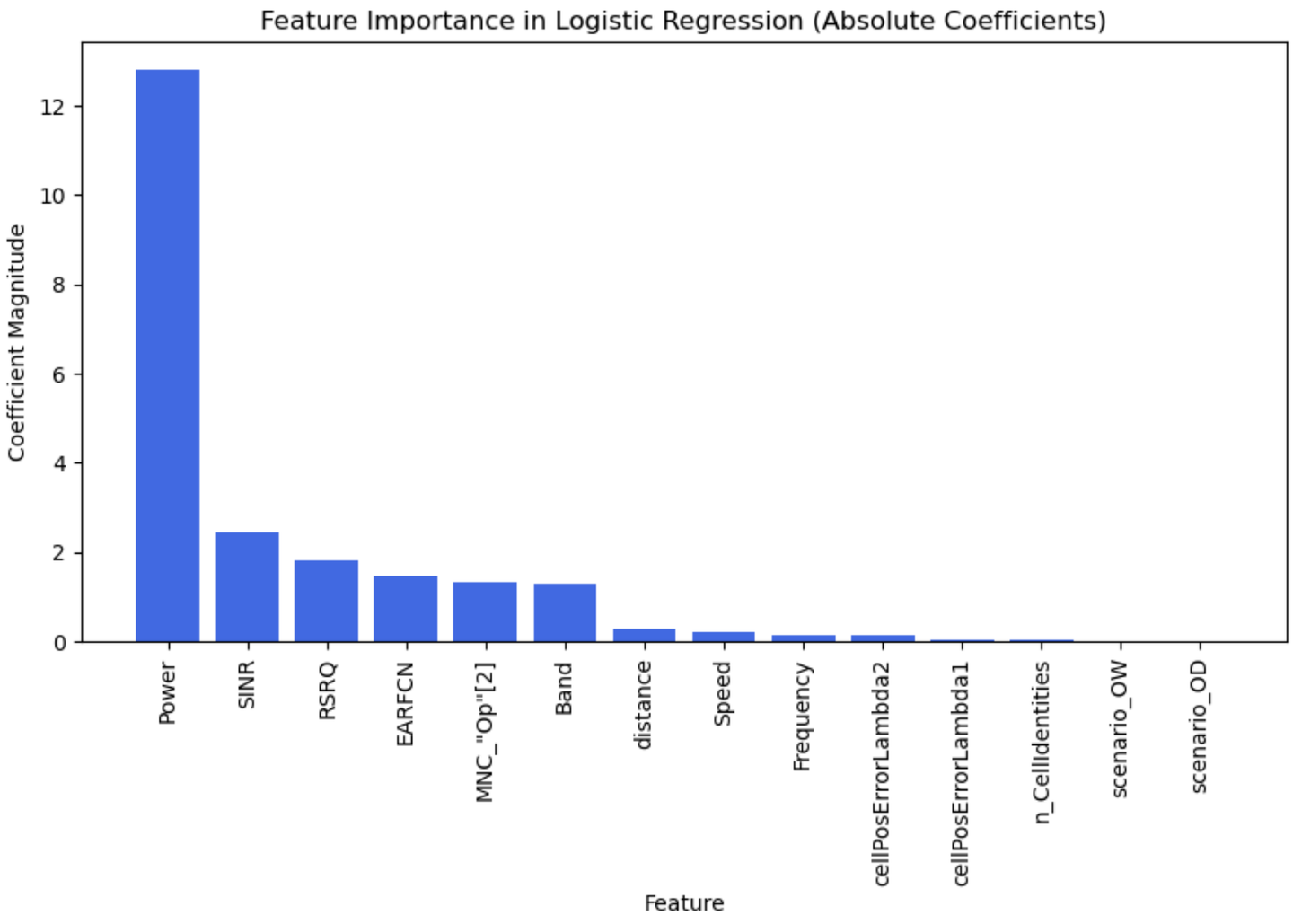
```
(32000, 14) (8000, 14)
```

In [86]:
```
# Get feature importance
feature_importances = rf_model.feature_importances_
sorted_idx = np.argsort(feature_importances)[::-1]

# Plot feature importances
plt.figure(figsize=(10, 5))
plt.bar(range(len(feature_importances)), feature_importances[sorted_idx], align="center")
plt.xticks(range(len(feature_importances)), X.columns[sorted_idx], rotation=90)
plt.xlabel("Feature")
plt.ylabel("Importance Score")
plt.title("Feature Importance in Random Forest")
plt.show()
```



In [87]:
```
# Get absolute coefficients from Logistic Regression model
logreg_coefficients = np.abs(log_reg.coef_[0])  # Extract coefficients

# Sort features by importance
sorted_idx = np.argsort(logreg_coefficients)[::-1]  # Sort in descending order
```

```
# Plot feature importance (absolute coefficient values)
plt.figure(figsize=(10, 5))
plt.bar(range(len(logreg_coefficients)), logreg_coefficients[sorted_idx], align="center", color="royalblue")
plt.xticks(range(len(logreg_coefficients)), X.columns[sorted_idx], rotation=90)
plt.xlabel("Feature")
plt.ylabel("Coefficient Magnitude")
plt.title("Feature Importance in Logistic Regression (Absolute Coefficients)")
plt.show()
```



Multi Classification

Feature Selection Criteria Use features where p-Value (Scenario) < 0.05 (statistically significant). Consider F-Statistic (Scenario) to prioritise stronger predictors.

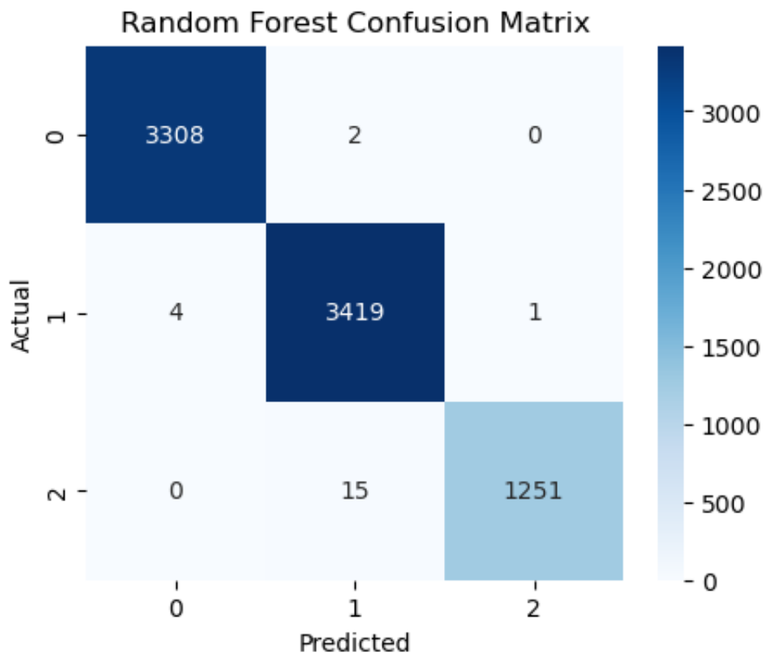
In [90]:
```
# Select significant features
selected_features = [
    "UTC", "Latitude", "Longitude", "Speed", "PCI", "Power",
    "RSRP", "cellLongitude", "cellLatitude", "Frequency",
    "Altitude", "distance", "MNC"
```

```
]

# Create a copy of the dataset
multi_classification = df_classification.copy()

# Keep only selected features + target variable
multi_classification = multi_classification[selected_features + ["scenario"]]

# Encode the target variable (Scenario)
multi_classification["scenario"] = multi_classification["scenario"].map({
    "IS": 0, "OW": 1, "OD": 2
})

# Encode 'MNC' (string to numeric)
le = LabelEncoder()
multi_classification["MNC"] = le.fit_transform(multi_classification["MNC"])

# Split into features and target
X = multi_classification.drop("scenario", axis=1)
y = multi_classification["scenario"]

# Train-test split
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42, stratify=y
)

# Standardize features
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

In [91]:

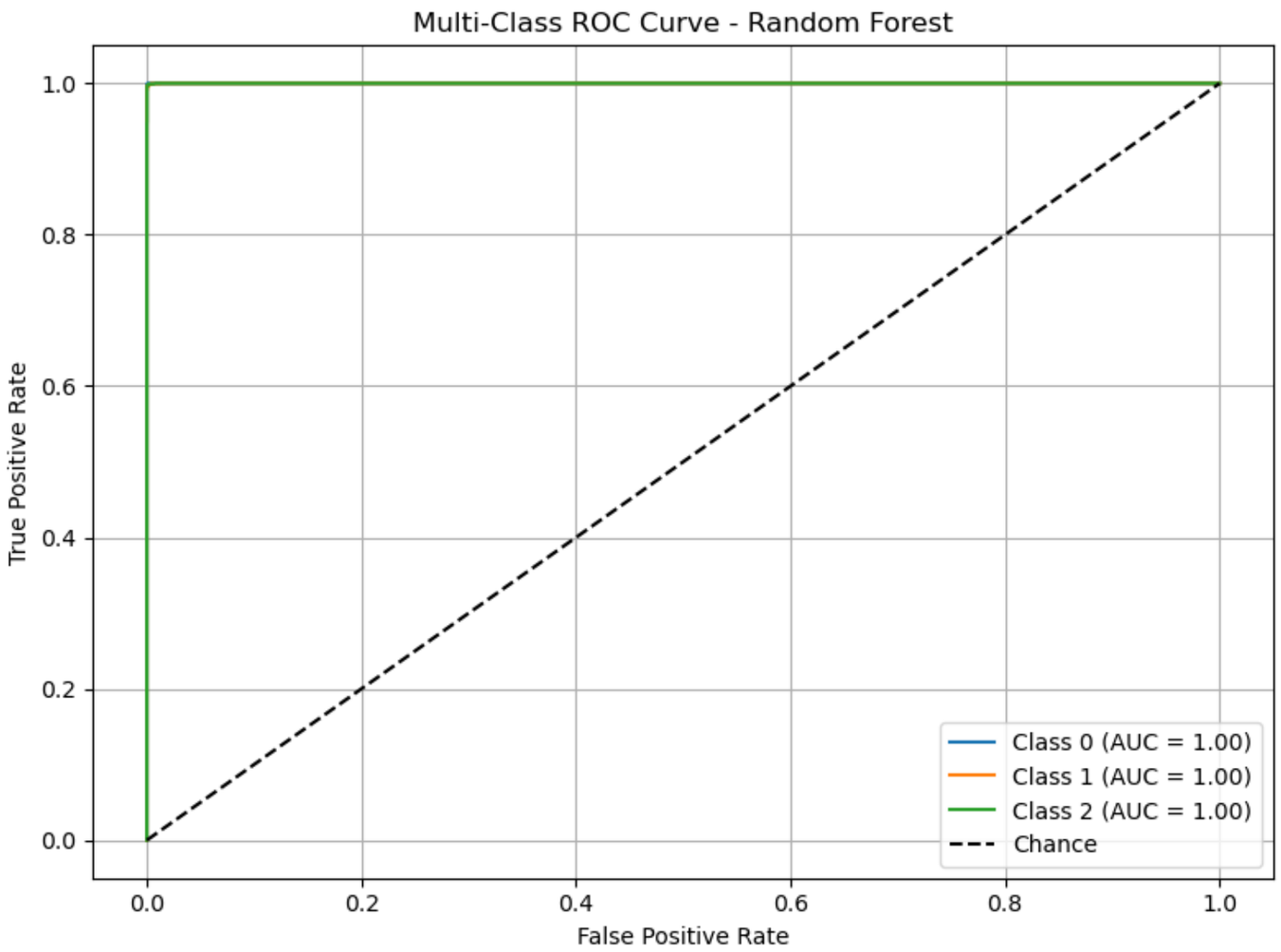
```
# Train a Random Forest Classifier
rf_clf = RandomForestClassifier(n_estimators=100, random_state=42)
rf_clf.fit(X_train_scaled, y_train)

# Predictions
y_pred_rf = rf_clf.predict(X_test_scaled)

# Evaluation
print("Random Forest Accuracy:", accuracy_score(y_test, y_pred_rf))
print(classification_report(y_test, y_pred_rf))

# Confusion Matrix
plt.figure(figsize=(5, 4))
sns.heatmap(confusion_matrix(y_test, y_pred_rf), annot=True, fmt="d", cmap="Blues")
plt.title("Random Forest Confusion Matrix")
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.show()
```

```
Random Forest Accuracy: 0.99725
              precision    recall  f1-score   support

           0       1.00      1.00      1.00      3310
           1       1.00      1.00      1.00      3424
           2       1.00      0.99      0.99      1266

    accuracy                           1.00      8000
   macro avg       1.00      1.00      1.00      8000
weighted avg       1.00      1.00      1.00      8000
```



In [92]:
```python
# Binarize the output
classes = np.unique(y_test)
y_test_bin = label_binarize(y_test, classes=classes)
n_classes = y_test_bin.shape[1]

# Get predicted probabilities
y_score = rf_clf.predict_proba(X_test_scaled)

# Compute ROC curve and AUC for each class
fpr = {}
tpr = {}
roc_auc = {}

for i in range(n_classes):
    fpr[i], tpr[i], _ = roc_curve(y_test_bin[:, i], y_score[:, i])
    roc_auc[i] = auc(fpr[i], tpr[i])

# Plot all ROC curves
plt.figure(figsize=(8, 6))
for i in range(n_classes):
    plt.plot(fpr[i], tpr[i], label=f"Class {classes[i]} (AUC = {roc_auc[i]:.2f})")
```

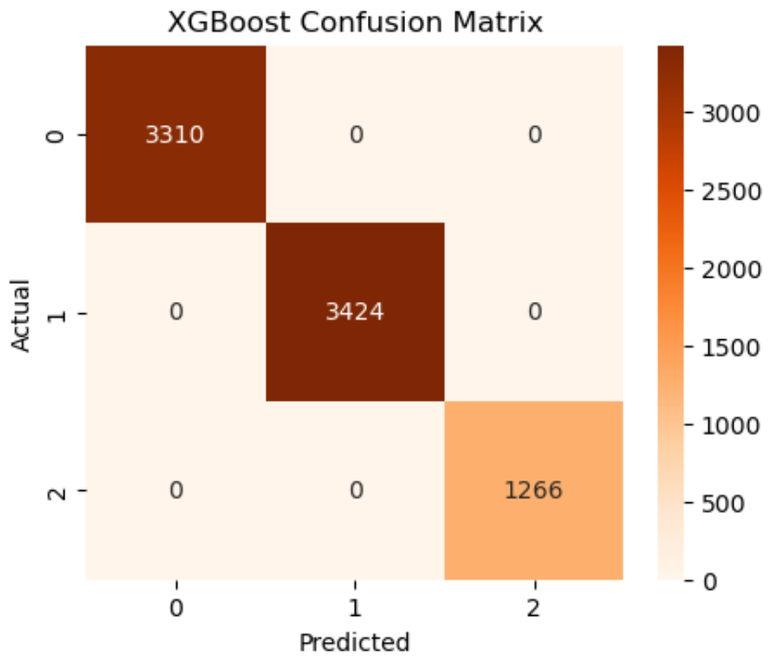
```
plt.plot([0, 1], [0, 1], 'k--', label='Chance')
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("Multi-Class ROC Curve - Random Forest")
plt.legend(loc="lower right")
plt.grid(True)
plt.tight_layout()
plt.show()
```



In [93]:
```
# Train an XGBoost Classifier for probability output
xgb_clf = XGBClassifier(objective="multi:softprob", num_class=3, eval_metric="mlogloss", random_state=42)
xgb_clf.fit(X_train_scaled, y_train)

# Predictions
y_pred_xgb = xgb_clf.predict(X_test_scaled)

# Evaluation
print("XGBoost Accuracy:", accuracy_score(y_test, y_pred_xgb))
print(classification_report(y_test, y_pred_xgb))
```

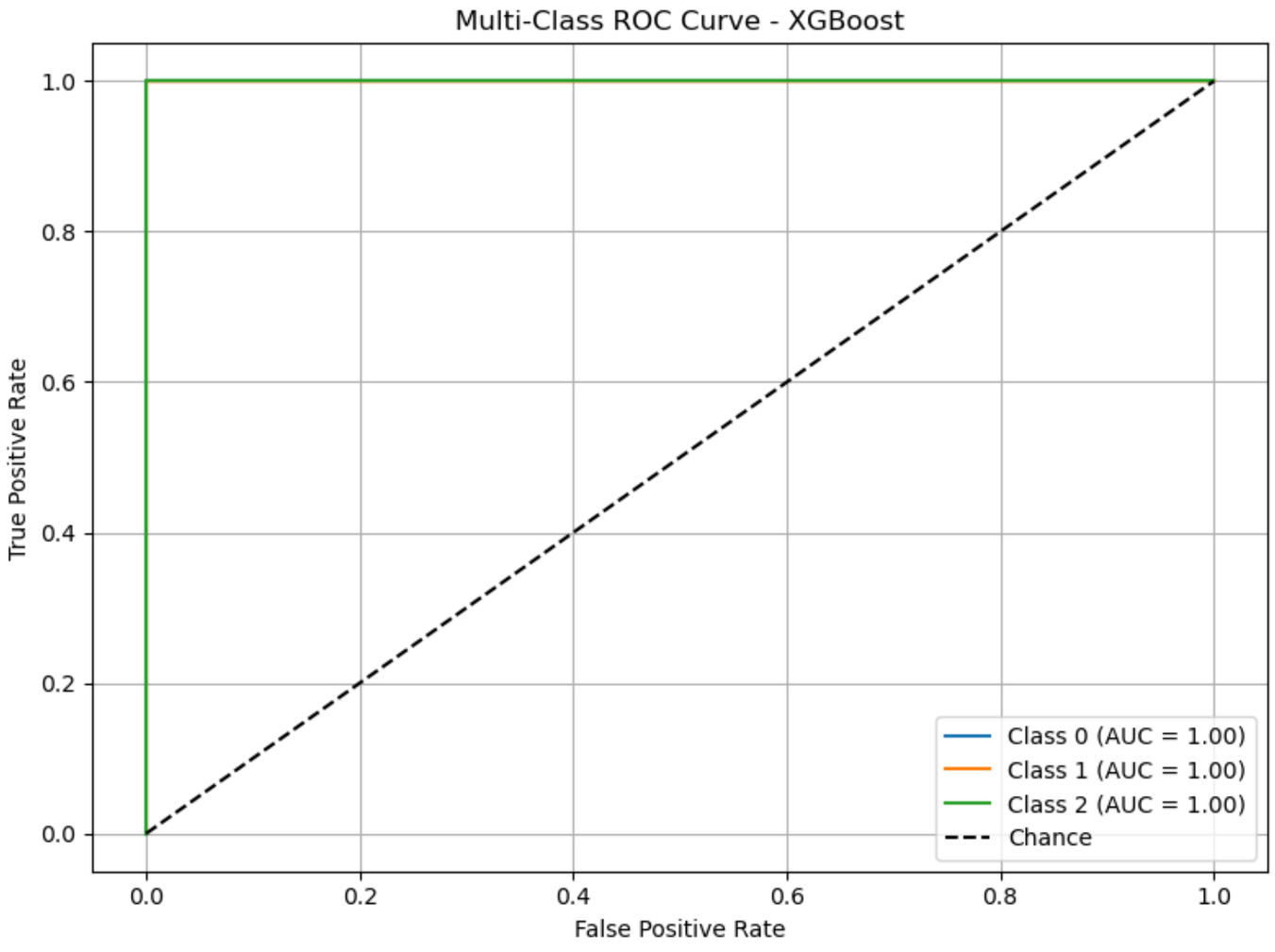
```
# Confusion Matrix
plt.figure(figsize=(5, 4))
sns.heatmap(confusion_matrix(y_test, y_pred_xgb), annot=True, fmt="d", cmap="Oranges")
plt.title("XGBoost Confusion Matrix")
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.show()
```

```
XGBoost Accuracy: 1.0
              precision    recall  f1-score   support

           0       1.00      1.00      1.00      3310
           1       1.00      1.00      1.00      3424
           2       1.00      1.00      1.00      1266

    accuracy                           1.00      8000
   macro avg       1.00      1.00      1.00      8000
weighted avg       1.00      1.00      1.00      8000
```



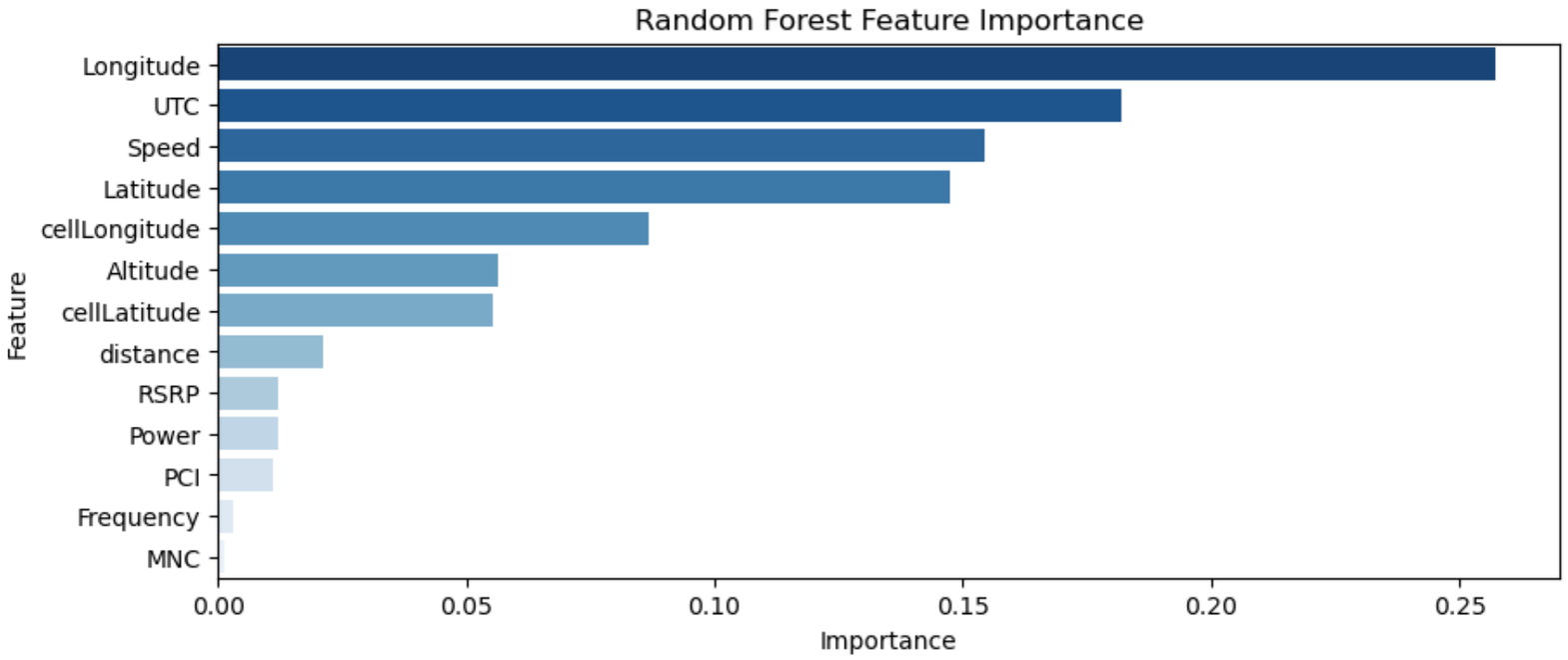
In [94]:
```
# Binarize the output
classes = np.unique(y_test)
y_test_bin = label_binarize(y_test, classes=classes)
n_classes = y_test_bin.shape[1]

# Get predicted probabilities
y_score = xgb_clf.predict_proba(X_test_scaled)

# Compute ROC curve and AUC for each class
fpr = {}
tpr = {}
roc_auc = {}
```

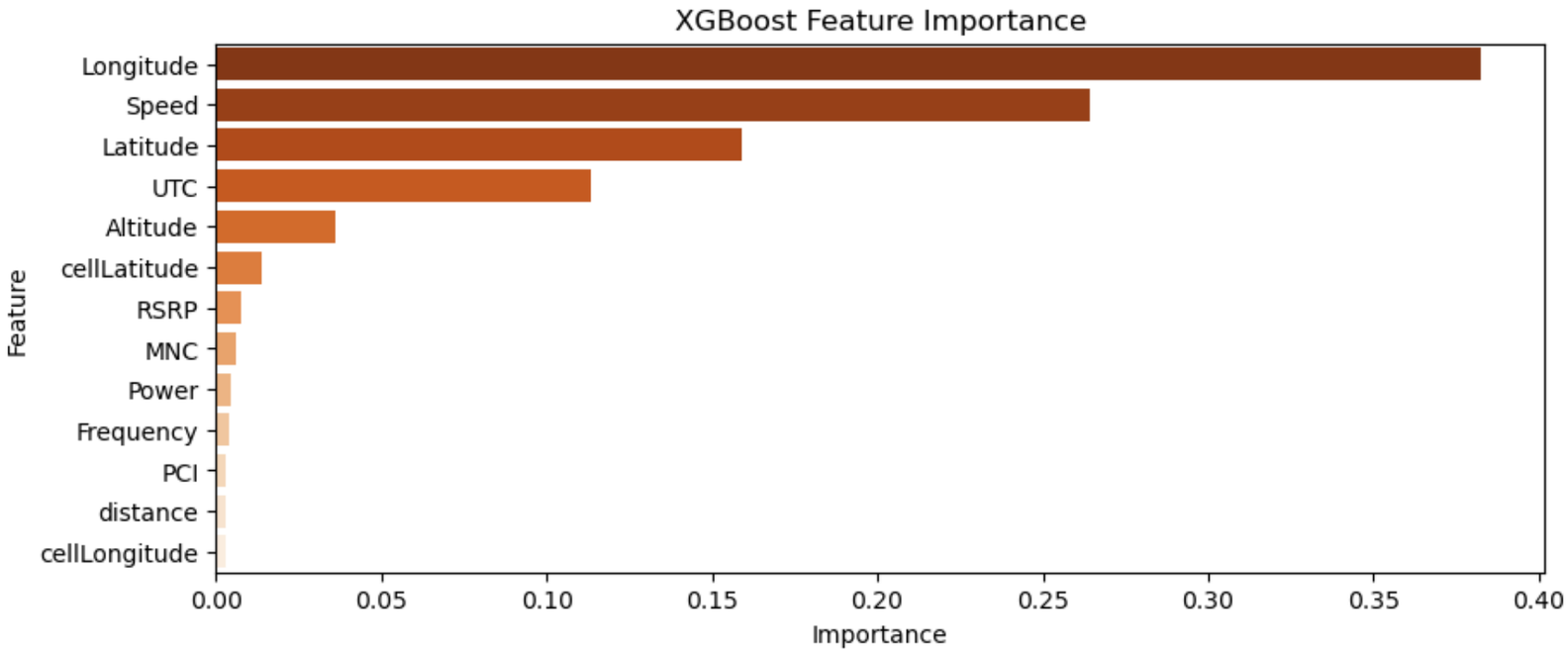
```
for i in range(n_classes):
    fpr[i], tpr[i], _ = roc_curve(y_test_bin[:, i], y_score[:, i])
    roc_auc[i] = auc(fpr[i], tpr[i])

# Plot ROC curves
plt.figure(figsize=(8, 6))
for i in range(n_classes):
    plt.plot(fpr[i], tpr[i], label=f"Class {classes[i]} (AUC = {roc_auc[i]:.2f})")

plt.plot([0, 1], [0, 1], 'k--', label='Chance')
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("Multi-Class ROC Curve - XGBoost")
plt.legend(loc="lower right")
plt.grid(True)
plt.tight_layout()
plt.show()
```



In [95]:
```
# Get feature importances from both models
rf_importance = pd.DataFrame({'Feature': selected_features, 'Importance': rf_clf.feature_importances_}).sort_values(by='Importance', ascending=False)
xgb_importance = pd.DataFrame({'Feature': selected_features, 'Importance': xgb_clf.feature_importances_}).sort_values(by='Importance', ascending=False)
```

```
# Plot Feature Importance - Random Forest
plt.figure(figsize=(10, 4))
sns.barplot(x='Importance', y='Feature', data=rf_importance, hue='Feature', legend=False, palette="Blues_r")
plt.title("Random Forest Feature Importance")
plt.show()

# Plot Feature Importance - XGBoost
plt.figure(figsize=(10, 4))
sns.barplot(x='Importance', y='Feature', data=xgb_importance, hue='Feature', legend=False, palette="Oranges_r")
plt.title("XGBoost Feature Importance")
plt.show()
```





TASK 4

Optimization Using Genetic Algorithms

In [97]:
```
# DATA PREPARATION
# Selecting relevant numerical features
features = ["distance", "RSRP", "Band", "Speed"]

# Standardizing the features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(df_clustered[features])

# FITNESS FUNCTION
def kmeans_silhouette(individual):
    """
    Runs K-Means with given hyperparameters (number of clusters k)
    and returns the silhouette score.
    """
    k = int(individual[0])  # Number of clusters

    try:
        kmeans = KMeans(n_clusters=k, random_state=42, n_init=10)
        labels = kmeans.fit_predict(X_scaled)

        # Ensure multiple clusters exist for silhouette score calculation
        if len(set(labels)) > 1:
            return silhouette_score(X_scaled, labels),
        else:
```

```
            return -1,  # Low score if only one cluster is found
    except:
        return -1,  # Return low score if algorithm fails

# GENETIC ALGORITHM SETUP
# Define optimization problem type (maximize silhouette score)
creator.create("FitnessMax", base.Fitness, weights=(1.0,))
creator.create("Individual", list, fitness=creator.FitnessMax)

# Define parameter range
BOUNDS_LOW = [2]   # Lower bound for k (min clusters)
BOUNDS_HIGH = [10]  # Upper bound for k (max clusters)

toolbox = base.Toolbox()
toolbox.register("attr_int", random.randint, BOUNDS_LOW[0], BOUNDS_HIGH[0])  # Cluster count k

# Create individuals
toolbox.register("individual", tools.initCycle, creator.Individual, (toolbox.attr_int,), n=1)
toolbox.register("population", tools.initRepeat, list, toolbox.individual)

# Register fitness function and genetic operators
toolbox.register("evaluate", kmeans_silhouette)
toolbox.register("mate", tools.cxUniform, indpb=0.5)  # Crossover
toolbox.register("mutate", tools.mutUniformInt, low=BOUNDS_LOW[0], up=BOUNDS_HIGH[0], indpb=0.2)  # Mutation
toolbox.register("select", tools.selTournament, tournsize=3)  # Selection

# RUN GENETIC ALGORITHM
def run_ga():
    population = toolbox.population(n=10)  # Population size
    hof = tools.HallOfFame(1)  # Store the best individual

    # Statistics
    stats = tools.Statistics(lambda ind: ind.fitness.values)
    stats.register("max", np.max)

    # Run Genetic Algorithm
    algorithms.eaSimple(population, toolbox, cxpb=0.8, mutpb=0.1, ngen=5,
                        stats=stats, halloffame=hof, verbose=True)

    return hof[0]  # Return best individual

# Run GA optimization
best_individual = run_ga()
best_k = int(best_individual[0])

print("\nBest Parameters Found by Genetic Algorithm:")
print(f"Optimal Number of Clusters (k): {best_k}")

# APPLY OPTIMIZED K-MEANS
# Run K-Means with best parameters found by GA
kmeans_optimized = KMeans(n_clusters=best_k, random_state=42, n_init=10)
labels_optimized = kmeans_optimized.fit_predict(X_scaled)

# Store the cluster labels in the original DataFrame
df_clustered["KMeans_Cluster_Optimized"] = labels_optimized
```

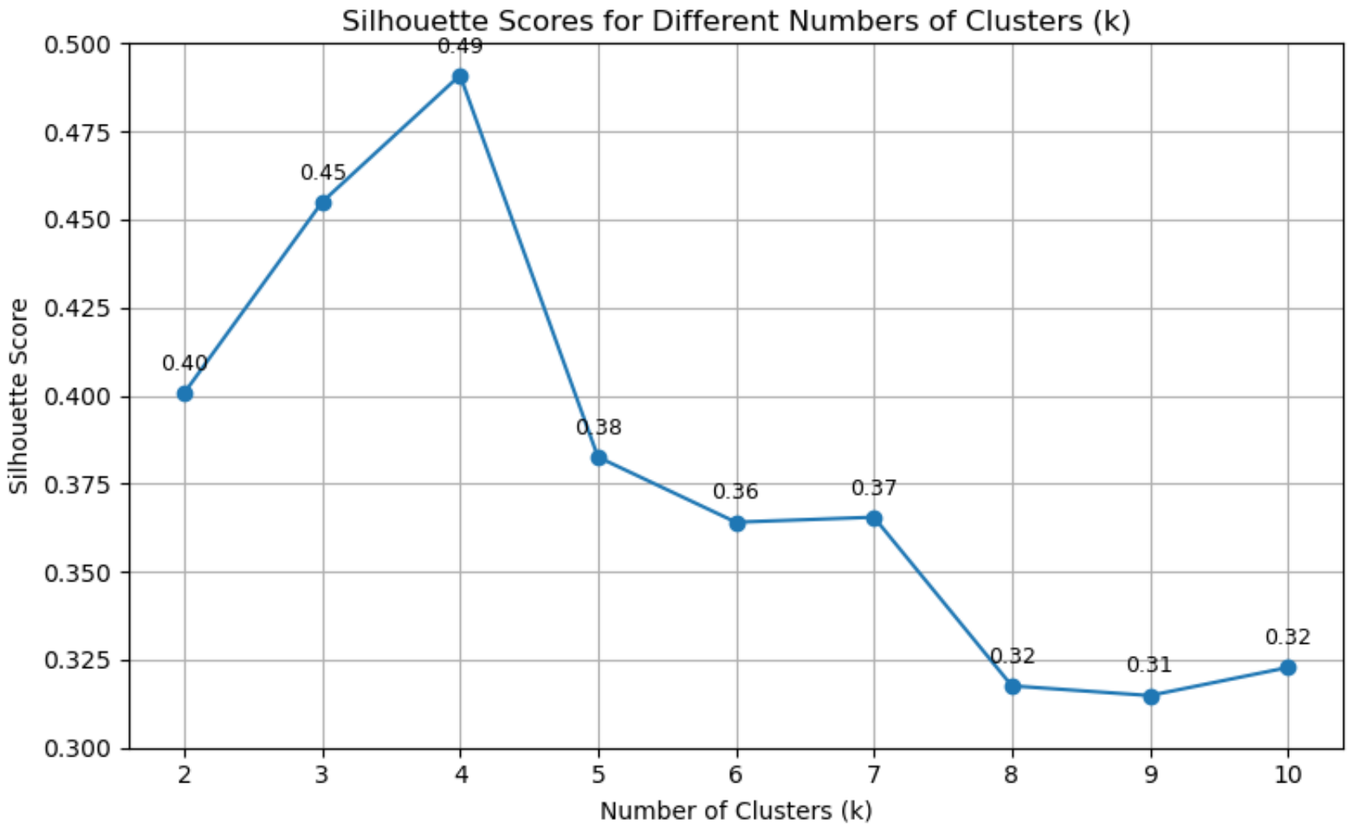
```
# Compute the silhouette score for the optimized clustering
silhouette_optimized = silhouette_score(X_scaled, labels_optimized)

print(f"\nNumber of clusters found: {best_k}")
print(f"Optimized Silhouette Score: {silhouette_optimized:.4f}")

# Display sample cluster counts
print("\nCluster Counts:")
print(pd.Series(labels_optimized).value_counts())

# Return optimized model and best parameters
kmeans_optimized, best_k, silhouette_optimized
```

```
gen     nevals  max
0       10      0.490957
1       10      0.490957
2       10      0.490957
3       8       0.490957
4       9       0.490957
5       8       0.490957

Best Parameters Found by Genetic Algorithm:
Optimal Number of Clusters (k): 4

Number of clusters found: 4
Optimized Silhouette Score: 0.4910

Cluster Counts:
0    26443
1     8816
3     2484
2     2257
Name: count, dtype: int64
```

Out[97]: (KMeans(n_clusters=4, n_init=10, random_state=42), 4, 0.4909566858263597)

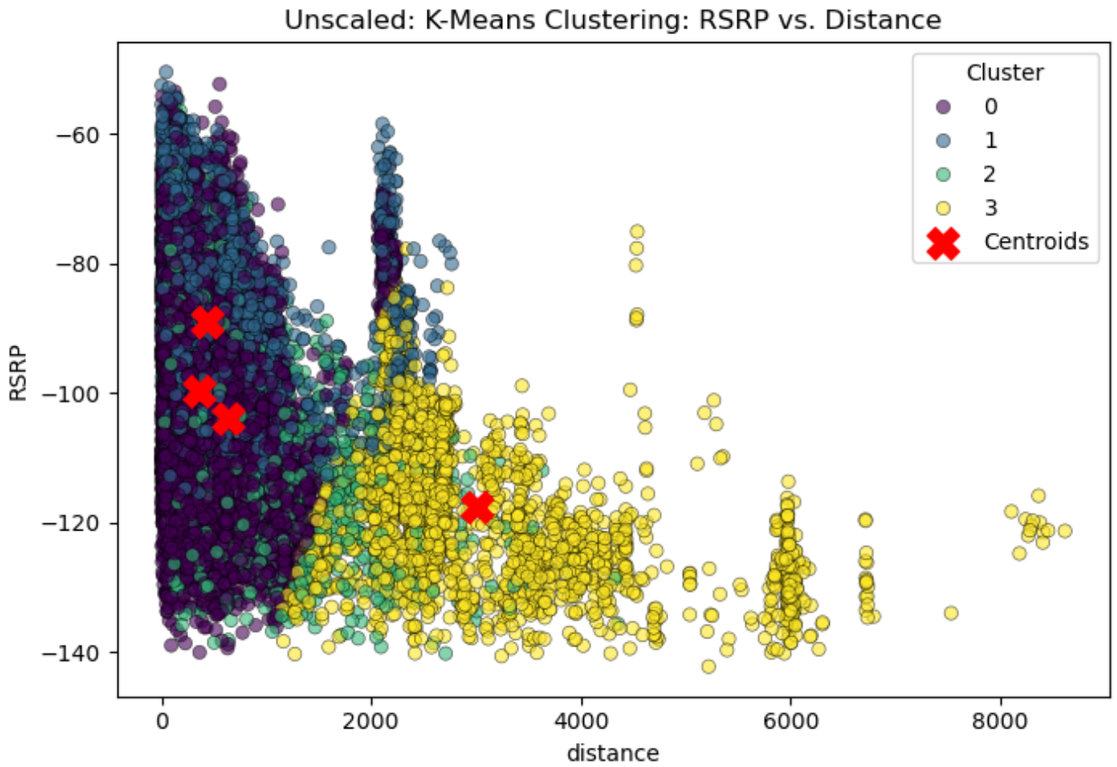
In [98]:
```
# Compute silhouette scores for different k values
silhouette_scores = []
k_range = range(2, 11)

for k in k_range:
    kmeans = KMeans(n_clusters=k, random_state=42, n_init=10)
    labels = kmeans.fit_predict(X_scaled)
    score = silhouette_score(X_scaled, labels)
    silhouette_scores.append(score)

# Plot silhouette scores with annotations
plt.figure(figsize=(8, 5))
plt.plot(k_range, silhouette_scores, marker='o', linestyle='-')

# Annotate score values above each point
for i, score in enumerate(silhouette_scores):
    plt.text(k_range[i], score + 0.005, f"{score:.2f}", ha='center', va='bottom', fontsize=9)

plt.title("Silhouette Scores for Different Numbers of Clusters (k)")
plt.xlabel("Number of Clusters (k)")
plt.ylabel("Silhouette Score")
```

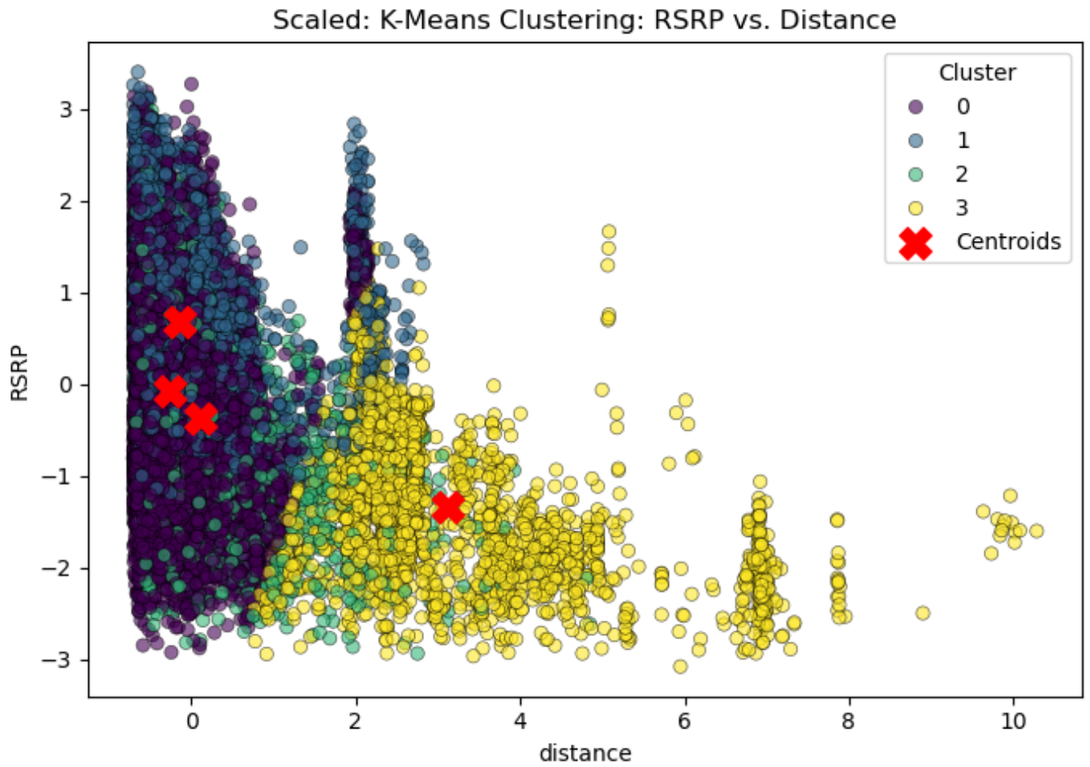
```
plt.grid(True)
plt.xticks(k_range)
plt.ylim(0.300, 0.500)  # Adjust Y-axis range
plt.tight_layout()
plt.show()
```

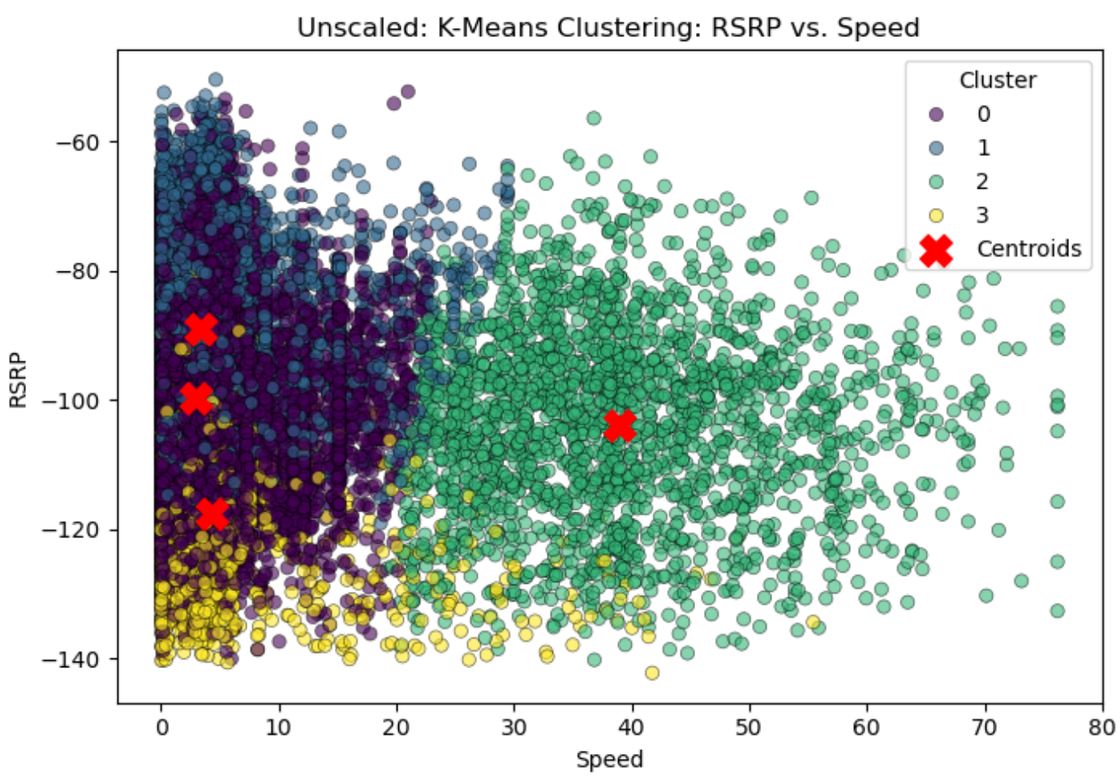


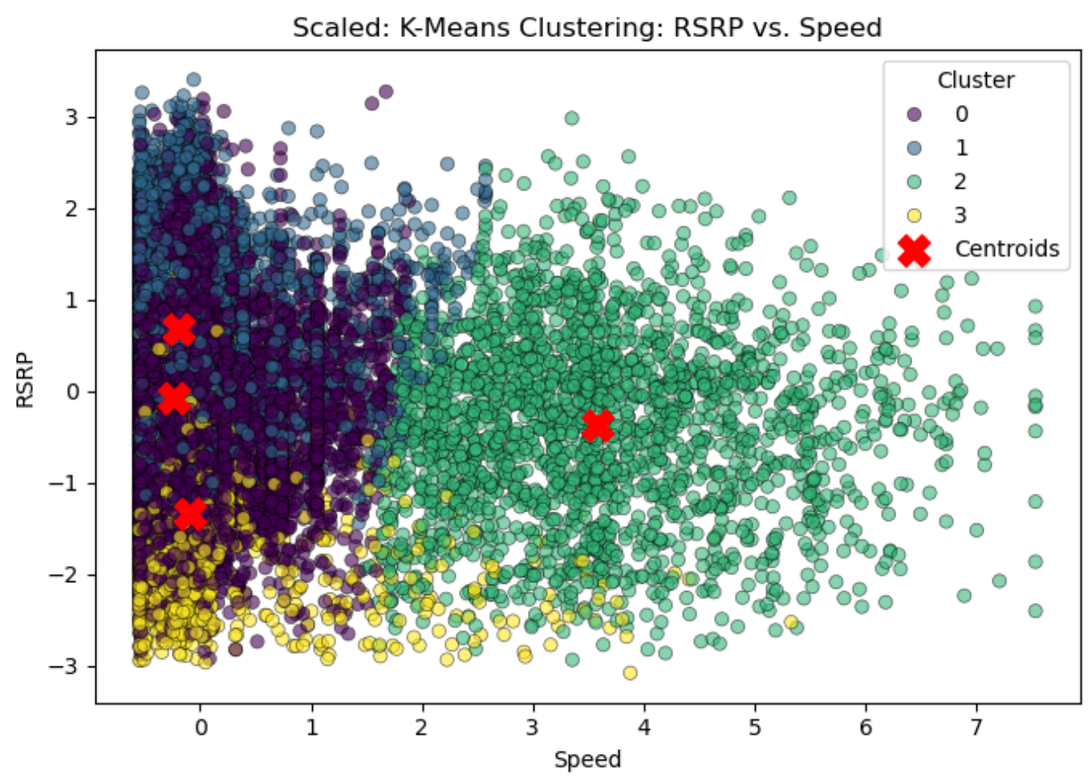
In [99]:
```
# Convert cluster centers back to original scale
centroids_scaled = kmeans_optimized.cluster_centers_  # Centroids in scaled space
centroids_original = scaler.inverse_transform(centroids_scaled)  # Convert back to original scale

# Convert `df_scaled` to DataFrame and add cluster labels
df_scaled = pd.DataFrame(X_scaled, columns=["distance", "RSRP", "Band", "Speed"])
df_scaled["KMeans_Cluster_Optimized"] = df_clustered["KMeans_Cluster_Optimized"].values  # Add cluster labels

# Define function to plot unscaled and scaled scatter plots with centroids
def plot_clusters_side_by_side(df_original, df_scaled, x_feature, y_feature, cluster_column, title, centroids_original=None, centroids_scaled=None):
    """
    Plots unscaled and scaled scatter plots side by side with centroids.

    Parameters:
        df_original (DataFrame): The original (unscaled) dataset.
        df_scaled (DataFrame): The scaled dataset.
        x_feature (str): Feature for x-axis.
        y_feature (str): Feature for y-axis.
        cluster_column (str): Column representing clusters.
        title (str): Title for the plots.
```

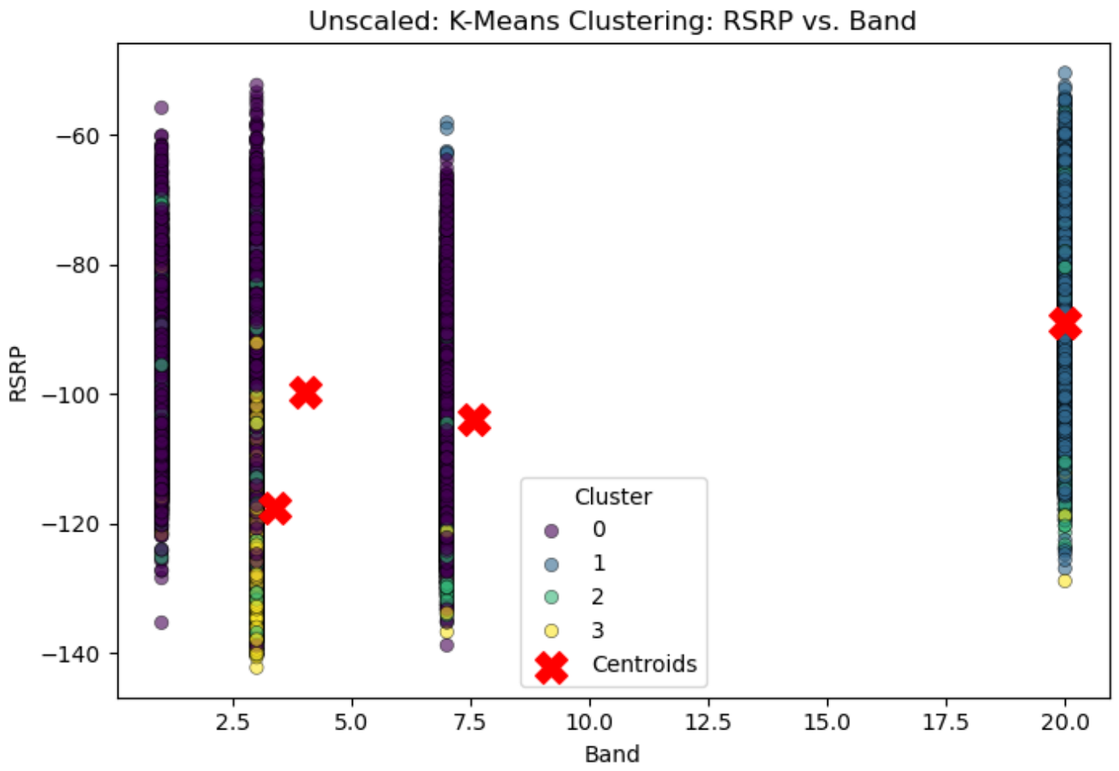
```
    centroids_original (array): Centroid positions in the original scale.
    centroids_scaled (array): Centroid positions in the scaled space.
"""
fig, axes = plt.subplots(1, 2, figsize=(14, 5))  # Side-by-side plots

# Unscaled Scatter Plot
sns.scatterplot(
    data=df_original, x=x_feature, y=y_feature, hue=cluster_column, palette='viridis', alpha=0.6, edgecolor='k', ax=axes[0]
)
if centroids_original is not None:
    axes[0].scatter(centroids_original[:, 0], centroids_original[:, 1], marker='X', s=200, color='red', label='Centroids')
axes[0].set_title(f"Unscaled: {title}")
axes[0].set_xlabel(x_feature)
axes[0].set_ylabel(y_feature)
axes[0].legend(title='Cluster')

# Scaled Scatter Plot
sns.scatterplot(
    data=df_scaled, x=x_feature, y=y_feature, hue=cluster_column, palette='viridis', alpha=0.6, edgecolor='k', ax=axes[1]
)
if centroids_scaled is not None:
    axes[1].scatter(centroids_scaled[:, 0], centroids_scaled[:, 1], marker='X', s=200, color='red', label='Centroids')
axes[1].set_title(f"Scaled: {title}")
axes[1].set_xlabel(x_feature)
axes[1].set_ylabel(y_feature)
axes[1].legend(title='Cluster')

plt.tight_layout()
    plt.show()

# Feature index mapping for centroids
distance_idx, rsrp_idx = 0, 1
speed_idx, band_idx = 3, 2

# Extract centroid positions for each plot
centroids_plot_distance = centroids_original[:, [distance_idx, rsrp_idx]]
centroids_scaled_distance = centroids_scaled[:, [distance_idx, rsrp_idx]]

centroids_plot_speed = centroids_original[:, [speed_idx, rsrp_idx]]
centroids_scaled_speed = centroids_scaled[:, [speed_idx, rsrp_idx]]

centroids_plot_band = centroids_original[:, [band_idx, rsrp_idx]]
centroids_scaled_band = centroids_scaled[:, [band_idx, rsrp_idx]]

# Plot Side-by-Side Comparisons for Each Feature Pair
plot_clusters_side_by_side(df_clustered, df_scaled, 'distance', 'RSRP', 'KMeans_Cluster_Optimized',
                           'K-Means Clustering: RSRP vs. Distance', centroids_plot_distance, centroids_scaled_distance)

plot_clusters_side_by_side(df_clustered, df_scaled, 'Speed', 'RSRP', 'KMeans_Cluster_Optimized',
                           'K-Means Clustering: RSRP vs. Speed', centroids_plot_speed, centroids_scaled_speed)

plot_clusters_side_by_side(df_clustered, df_scaled, 'Band', 'RSRP', 'KMeans_Cluster_Optimized',
                           'K-Means Clustering: RSRP vs. Band', centroids_plot_band, centroids_scaled_band)
```

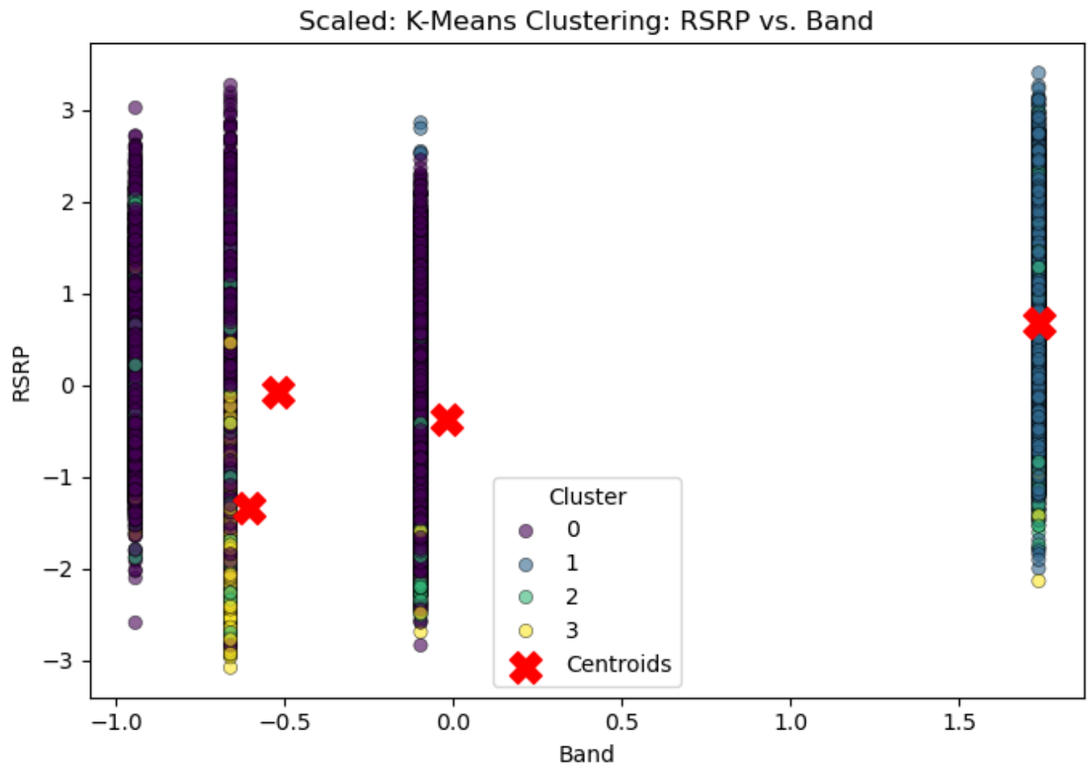

Unscaled: K-Means Clustering: RSRP vs. Distance
RSRP
distance
Cluster
0
1
2
3
Centroids
−60
−80
−100
−120
−140
0
2000
4000
6000
8000


Scaled: K-Means Clustering: RSRP vs. Distance
RSRP
distance
Cluster
0
1
2
3
Centroids
3
2
1
0
−1
−2
−3
0
2
4
6
8
10


Unscaled: K-Means Clustering: RSRP vs. Speed
RSRP
Speed
Cluster
0
1
2
3
Centroids
−60
−80
−100
−120
−140
0
10
20
30
40
50
60
70
80


Scaled: K-Means Clustering: RSRP vs. Speed
RSRP
Speed
Cluster
0
1
2
3
Centroids
3
2
1
0
−1
−2
−3
0
1
2
3
4
5
6
7





In [100…

```python
# DATA PREPARATION
# Define features and target variable
X = df_binary_classification.drop(columns=["RSRP", "RSRP_Class"])  # Features
y = df_binary_classification["RSRP_Class"]  # Target variable

# Split into training and testing sets (80/20 split, stratified)
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, stratify=y, random_state=42
)

# Scale numeric features (Random Forest does not require scaling, but keeping for consistency)
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# BASELINE MODEL
baseline_model = RandomForestClassifier(random_state=42, n_jobs=-1)
baseline_model.fit(X_train, y_train)
y_pred_baseline = baseline_model.predict(X_test)

# Evaluate baseline model
baseline_metrics = {
    "Accuracy": accuracy_score(y_test, y_pred_baseline),
    "Precision": precision_score(y_test, y_pred_baseline),
    "Recall": recall_score(y_test, y_pred_baseline),
    "F1 Score": f1_score(y_test, y_pred_baseline)
}
```

```
print("\nBaseline Model Classification Report:")
print(classification_report(y_test, y_pred_baseline))

# GENETIC ALGORITHM OPTIMIZATION
# Define search space for Genetic Algorithm tuning
param_grid = {
    "n_estimators": Integer(50, 500),  # Number of trees in the forest
    "max_depth": Integer(3, 50),  # Depth of the trees
    "min_samples_split": Integer(2, 10),  # Minimum number of samples required to split
    "min_samples_leaf": Integer(1, 5),  # Minimum samples per leaf
    "criterion": Categorical(["gini", "entropy"])  # Splitting criteria
}

# Set up Genetic Algorithm Search with 3-fold CV
ga_search = GASearchCV(
    estimator=RandomForestClassifier(random_state=42, n_jobs=-1),
    param_grid=param_grid,
    cv=3,
    scoring="accuracy",
    population_size=20,
    generations=10,
    mutation_probability=0.1,
    crossover_probability=0.8,
    n_jobs=-1,
    verbose=True
)
```

```
# Run Genetic Algorithm optimization
ga_search.fit(X_train, y_train)
```

```
Baseline Model Classification Report:
              precision    recall  f1-score   support

           0       0.97      0.97      0.97      3653
           1       0.97      0.97      0.97      4347

    accuracy                           0.97      8000
   macro avg       0.97      0.97      0.97      8000
weighted avg       0.97      0.97      0.97      8000
```

```
/opt/anaconda3/lib/python3.12/site-packages/deap/creator.py:185: RuntimeWarning: A class named 'FitnessMax' has already been created and it will be overwritten. Co
nsider deleting previous creation of that class or rename it.
  warnings.warn("A class named '{0}' has already been created and it "
/opt/anaconda3/lib/python3.12/site-packages/deap/creator.py:185: RuntimeWarning: A class named 'Individual' has already been created and it will be overwritten. Co
nsider deleting previous creation of that class or rename it.
  warnings.warn("A class named '{0}' has already been created and it "
```

gen	nevals	fitness	fitness_std	fitness_max	fitness_min
0	20	0.966147	0.00895486	0.970406	0.937531
1	29	0.96997	0.000384951	0.970688	0.968938
2	36	0.970234	0.000332847	0.971156	0.969594
3	36	0.970466	0.000274018	0.971156	0.969969
4	36	0.970692	0.000273195	0.971156	0.9705
5	38	0.970905	0.000352512	0.971406	0.9705
6	39	0.971128	0.000289279	0.971406	0.9705
7	34	0.971255	0.000291363	0.971406	0.970156
8	35	0.971394	5.44896e-05	0.971406	0.971156
9	30	0.971406	0	0.971406	0.971406
10	39	0.971406	0	0.971406	0.971406

Out[100…

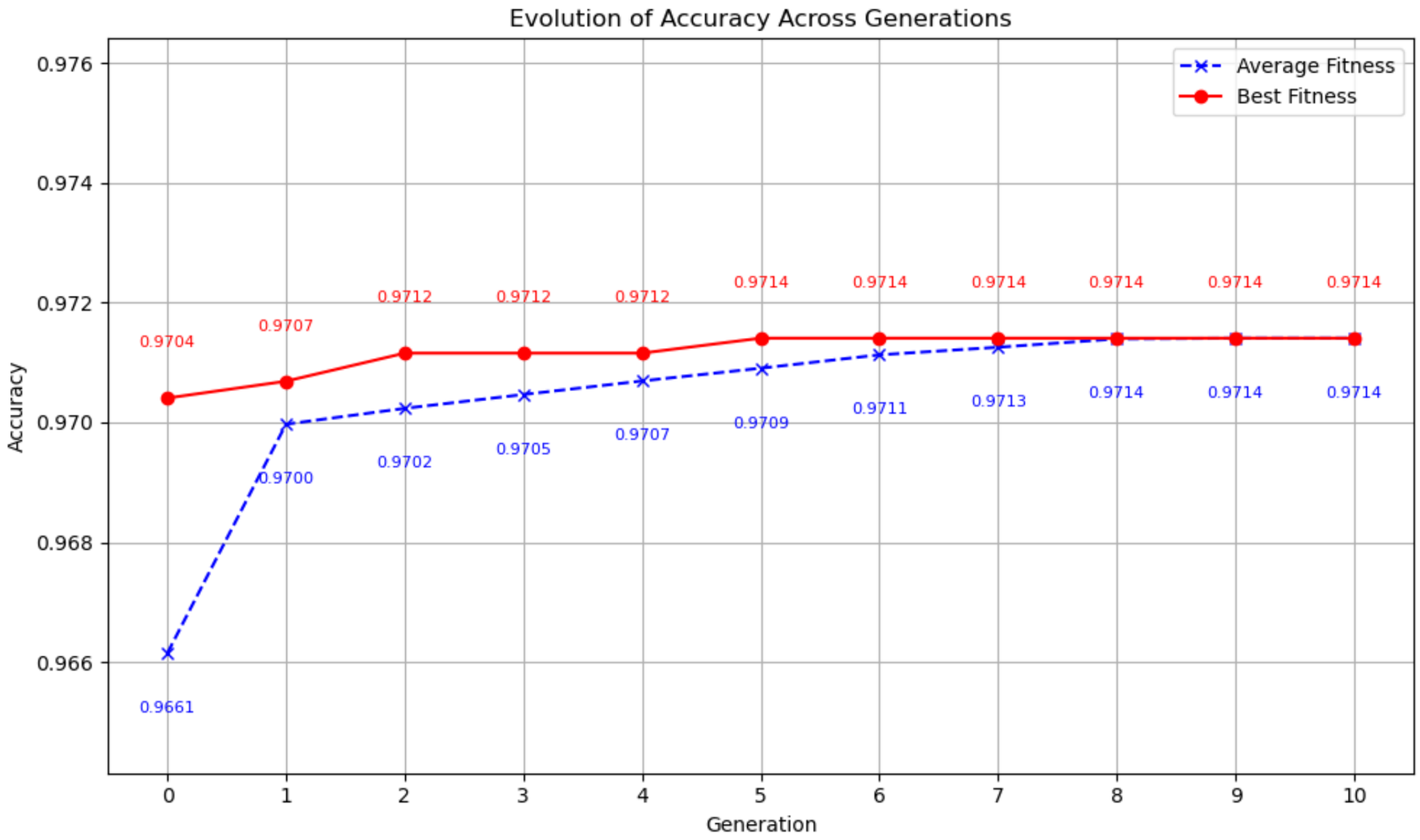
GASearchCV

estimator: RandomForestClassifier

RandomForestClassifier

In [101…

```python
# Extract values
generations = ga_search.history['gen']
avg_scores = ga_search.history['fitness']
best_scores = ga_search.history['fitness_max']

# Plot both average and best fitness
plt.figure(figsize=(10, 6))
plt.plot(generations, avg_scores, marker='x', linestyle='--', label='Average Fitness', color='blue')
plt.plot(generations, best_scores, marker='o', linestyle='-', label='Best Fitness', color='red')
plt.title("Evolution of Accuracy Across Generations")
plt.xlabel("Generation")
plt.ylabel("Accuracy")
plt.grid(True)

# Annotate average fitness values
for i, score in enumerate(avg_scores):
    plt.text(generations[i], score - 0.0008, f"{score:.4f}", ha='center', va='top', fontsize=8, color='blue')

# Annotate best fitness values
for i, score in enumerate(best_scores):
    plt.text(generations[i], score + 0.0008, f"{score:.4f}", ha='center', va='bottom', fontsize=8, color='red')

plt.xticks(generations)
plt.ylim(min(avg_scores) - 0.002, max(best_scores) + 0.005)  # Add padding
plt.legend()
plt.tight_layout()
plt.show()
```



In [102…

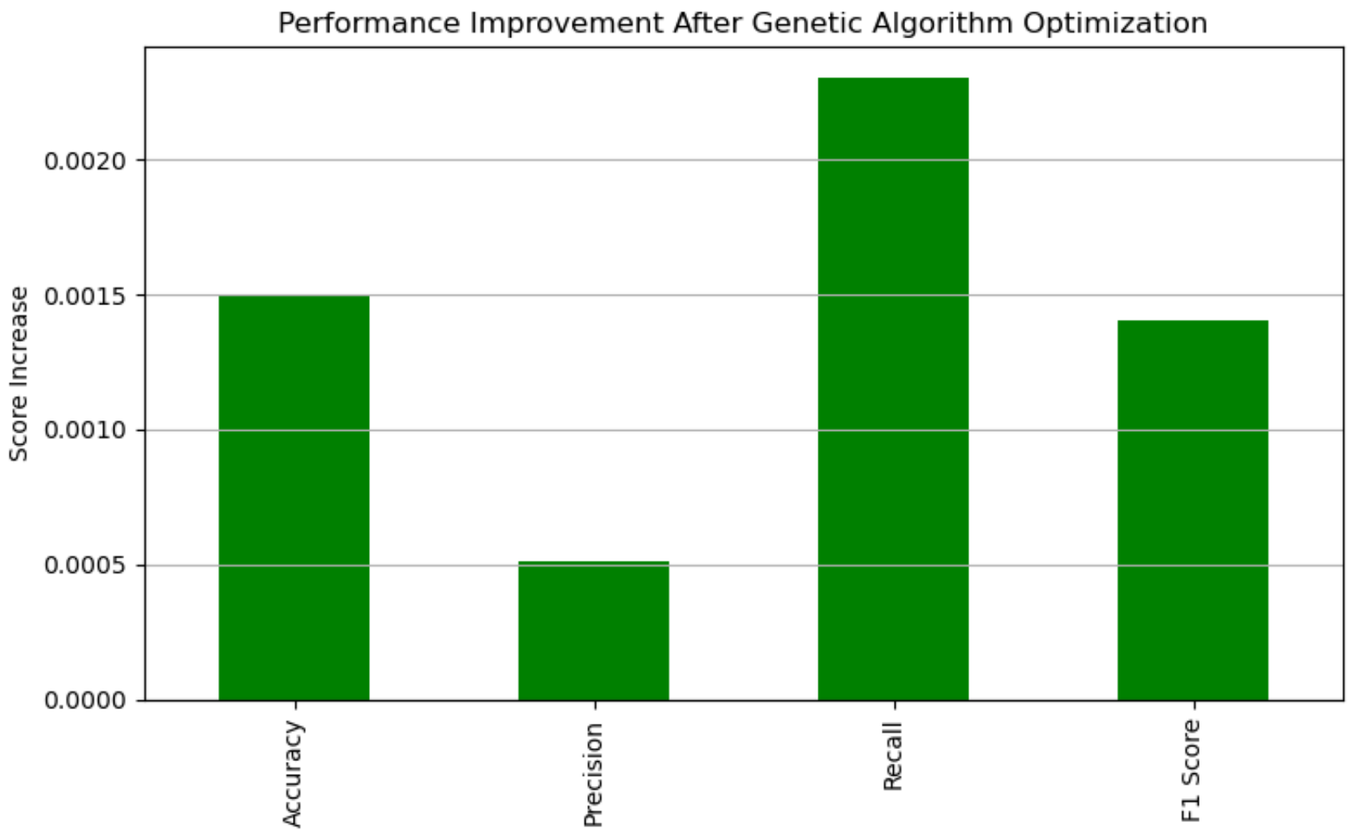
```python
# OPTIMIZED MODEL
# Train Random Forest with best GA parameters
best_params = ga_search.best_params_
optimized_model = RandomForestClassifier(
    **best_params, random_state=42, n_jobs=-1
)
optimized_model.fit(X_train, y_train)
y_pred_optimized = optimized_model.predict(X_test)

# Evaluate optimized model
optimized_metrics = {
    "Accuracy": accuracy_score(y_test, y_pred_optimized),
    "Precision": precision_score(y_test, y_pred_optimized),
    "Recall": recall_score(y_test, y_pred_optimized),
    "F1 Score": f1_score(y_test, y_pred_optimized)
}

print("\nOptimized Model Classification Report:")
print(classification_report(y_test, y_pred_optimized))
```

```
# Store results
results_df = pd.DataFrame([baseline_metrics, optimized_metrics], index=['Baseline', 'Optimized'])

# Display results in a readable format
print("\nModel Performance Comparison:")
print(results_df)

# Print best hyperparameters found by GA
print("\nBest Hyperparameters Found by Genetic Algorithm:")
print(best_params)
```

```
Optimized Model Classification Report:
              precision    recall  f1-score   support

           0       0.97      0.97      0.97      3653
           1       0.97      0.98      0.97      4347

    accuracy                           0.97      8000
   macro avg       0.97      0.97      0.97      8000
weighted avg       0.97      0.97      0.97      8000


Model Performance Comparison:
           Accuracy  Precision    Recall  F1 Score
Baseline    0.97075   0.971999  0.974235  0.973116
Optimized   0.97225   0.972509  0.976536  0.974518

Best Hyperparameters Found by Genetic Algorithm:
{'n_estimators': 163, 'max_depth': 48, 'min_samples_split': 2, 'min_samples_leaf': 1, 'criterion': 'entropy'}
```

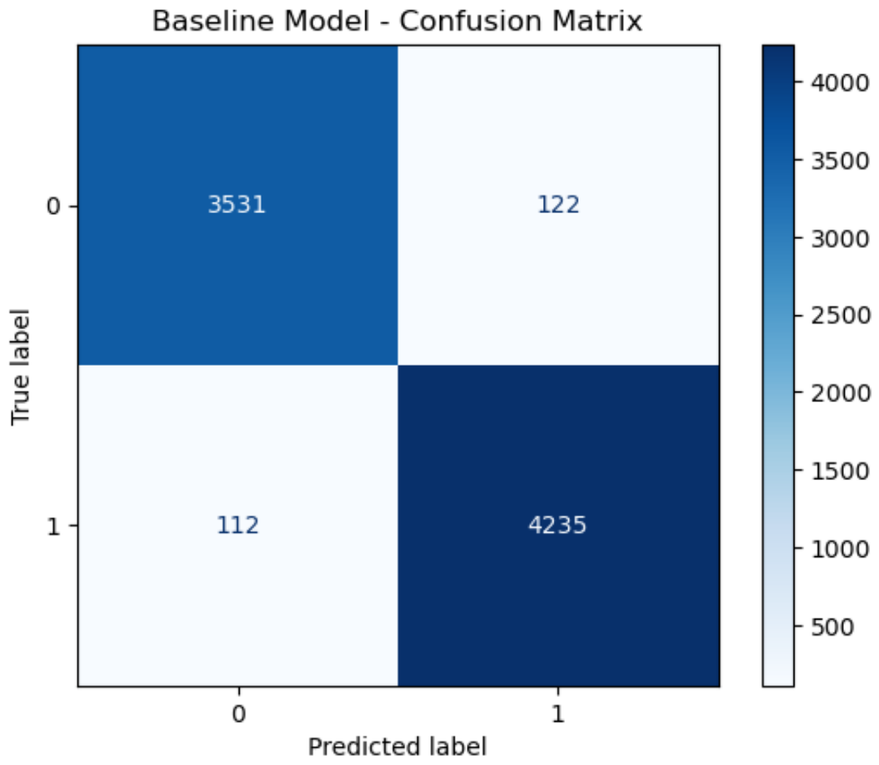
In [103…

```
improvement = results_df.loc['Optimized'] - results_df.loc['Baseline']

# Plot the improvements only
improvement.plot(kind='bar', color='green', figsize=(8, 5))
plt.title("Performance Improvement After Genetic Algorithm Optimization")
plt.ylabel("Score Increase")
plt.grid(axis='y')
plt.axhline(0, color='black', linewidth=0.8)
plt.tight_layout()
plt.show()
```



In [104…

```
# Confusion matrix - baseline
ConfusionMatrixDisplay.from_estimator(
    baseline_model, X_test, y_test, cmap="Blues", values_format='d'
)
plt.title("Baseline Model - Confusion Matrix")
plt.show()

# Confusion matrix - optimized
ConfusionMatrixDisplay.from_estimator(
    optimized_model, X_test, y_test, cmap="Greens", values_format='d'
)
plt.title("Optimized Model - Confusion Matrix")
plt.show()
```



Optimized Model - Confusion Matrix