Laboratory Tutorial: GPS NLOS Signal Identification Using Machine Learning

Dr. Weisong Wen
Department of Aeronautical and Aviation Engineering
The Hong Kong Polytechnic University

March 4, 2025

Contents

1 Introduction and Background

The Global Positioning System (GPS) and other Global Navigation Satellite Systems (GNSS) have become integral to modern navigation and positioning applications. In ideal scenarios, GPS receivers capture signals that travel along direct Line-of-Sight (LOS) paths from satellites. However, in urban environments, indoor spaces, or areas with dense vegetation, signals often reach receivers after reflecting off surrounding surfaces–a phenomenon known as Non-Line-of-Sight (NLOS) reception.

NLOS signals introduce significant positioning errors because:

- They travel longer distances before reaching the receiver, introducing positive pseudorange errors
- They experience power attenuation due to reflection and diffraction
- They may cause multipath interference when combined with direct signals

Differentiating between LOS and NLOS signals is crucial for accurate positioning, especially in challenging environments. Machine learning techniques offer promising approaches to identify and mitigate NLOS signals based on observable signal characteristics.

1.1 Key Features for NLOS Detection

This laboratory exercise focuses on four primary features used to distinguish between LOS and NLOS signals:

Table 1: Typical Values of GPS Signal Features for LOS and NLOS Conditions

Feature	**LOS**	**NLOS**
Signal-to-Noise Ratio (SNR)	35-55 dB-Hz	15-35 dB-Hz
Elevation Angle	30-90°	5-30°
Azimuth Angle	Any (0-360°)	Building-dependent
Constellation	GPS, GLONASS, Galileo, BeiDou	
Mean SNR	40.8 dB-Hz	28.5 dB-Hz
Mean Elevation	49.2°	21.6°
Pseudorange Error	0-5 m	5-100+ m

1.1.1 Signal-to-Noise Ratio (SNR)

SNR measures the power ratio between the desired signal and background noise, expressed in decibels (dB). NLOS signals typically exhibit lower SNR values because:

- Signal power decreases when reflected off surfaces
- Reflection introduces phase shifts and distortion
- Multiple reflections further attenuate signal strength

In general, a higher SNR value (typically $> 35 - 40$ dB-Hz for GPS signals) suggests a higher probability of LOS conditions, while lower values may indicate NLOS reception.

1.1.2 Satellite Constellation

Different satellite constellations (GPS, GLONASS, Galileo, BeiDou) operate on different frequency bands with varying signal characteristics:

- Signal wavelengths affect how signals interact with obstacles
- Signal modulation techniques influence resistance to multipath
- Satellite orbit geometry impacts typical elevation angles

The constellation parameter helps account for these systematic differences when building detection models.

1.1.3 Elevation Angle

Elevation angle represents the vertical angle between the satellite and the horizon from the receiver's perspective, measured in degrees:

- High-elevation satellites (e.g., $> 30°$) are more likely to have clear LOS paths
- Low-elevation satellites are more susceptible to obstruction by buildings, terrain, and vegetation
- Signals from low-elevation satellites travel through more atmosphere, reducing SNR

Elevation angle is a strong indicator of NLOS probability–signals from satellites near the horizon have a higher likelihood of being NLOS in urban environments.

1.1.4 Azimuth Angle

Azimuth angle represents the horizontal direction from which the satellite signal arrives, measured clockwise from north in degrees (0-360°):

- In urban environments, buildings predominantly block signals from specific directions
- When combined with 3D city models, azimuth can strongly predict NLOS probability
- Regular patterns in azimuth-related NLOS events can reveal environmental characteristics

Azimuth angle allows the detection model to account for directional patterns in signal obstruction.

2 Machine Learning Methods for NLOS Detection

This lab introduces four machine learning techniques for NLOS signal classification:

2.1 Linear Regression

2.1.1 Theoretical Background

Linear regression models the relationship between a dependent variable y (in our case, a continuous measure related to NLOS probability) and independent variables X (our four features) using a linear equation:

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3 + \beta_4 x_4 + \epsilon \tag{1}$$

Where:

- y is the predicted value
- β_0 is the intercept
- $\beta_1, \beta_2, \beta_3, \beta_4$ are coefficients for each feature
- x_1, x_2, x_3, x_4 represent our features (SNR, Constellation, Elevation, Azimuth)

- ϵ is the error term

The coefficients are determined by minimizing the sum of squared residuals:

$$\min_{\beta} \sum_{i=1}^{n} (y_i - (\beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + \beta_3 x_{i3} + \beta_4 x_{i4}))^2 \tag{2}$$

2.1.2 Application to NLOS Detection

While linear regression typically predicts continuous values rather than binary classes, we can:

- Use linear regression to predict a pseudo-probability score
- Apply a threshold to classify signals as LOS or NLOS
- Analyze coefficients to understand feature importance

The limitations of linear regression include:

- Assumes linear relationships between features and target
- May produce predictions outside the [0,1] range, requiring normalization
- Often outperformed by dedicated classification algorithms for binary problems

2.2 Logistic Regression

2.2.1 Theoretical Background

Logistic regression extends linear regression to binary classification by applying the logistic function to the linear predictor:

$$P(y = 1|X) = \frac{1}{1 + e^{-(\beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3 + \beta_4 x_4)}} \tag{3}$$

The model outputs a probability between 0 and 1, which can be interpreted as the probability of the signal being NLOS. The decision boundary is typically set at 0.5, classifying signals with predicted probabilities above this threshold as NLOS.

The logistic regression coefficients are estimated using maximum likelihood estimation rather than least squares:

$$\max_{\beta} \sum_{i=1}^{n} [y_i \log(p_i) + (1 - y_i) \log(1 - p_i)] \tag{4}$$

Where p_i is the predicted probability for the i-th observation.

2.2.2 Application to NLOS Detection

Logistic regression offers several advantages for NLOS detection:

- Directly outputs probabilities, allowing for confidence assessment
- More appropriate for binary classification than linear regression
- Coefficients provide interpretable feature importance
- Computationally efficient for real-time applications

The logistic function transforms the linear combination of features into a probability score, making it well-suited for classification tasks where the relationship between features and class probability is roughly sigmoidal.

2.3 Decision Trees

2.3.1 Theoretical Background

Decision trees recursively partition the feature space into regions, making decisions at each node based on feature thresholds:

- Root node: Contains the entire dataset
- Internal nodes: Split the data based on feature thresholds
- Leaf nodes: Contain the final predictions

The algorithm selects splits that maximize information gain or minimize impurity (typically measured using Gini impurity or entropy). For a binary classification like NLOS detection, Gini impurity at node m is:

$$G_m = 1 - \sum_{k=1}^{K} p_{mk}^2 = 2 p_{m,\text{LOS}} p_{m,\text{NLOS}} \tag{5}$$

Where p_{mk} is the proportion of class k observations in node m.

The decision tree algorithm recursively selects the feature and threshold that result in the largest reduction in impurity:

$$\Delta G = G_{\text{parent}} - \frac{N_{\text{left}}}{N_{\text{parent}}} G_{\text{left}} - \frac{N_{\text{right}}}{N_{\text{parent}}} G_{\text{right}} \tag{6}$$

2.3.2 Application to NLOS Detection

Decision trees offer several benefits for NLOS detection:

- Can capture non-linear relationships between features and NLOS probability
- Easily interpretable, showing clear decision rules
- Handle mixed feature types (continuous and categorical) naturally
- Require minimal preprocessing of data

For example, a decision tree might create rules like:

- If SNR < 30 dB-Hz AND Elevation < 15, classify as NLOS
- If SNR ≥ 40 dB-Hz AND Elevation ≥ 45, classify as LOS

These clear decision boundaries make the classification process transparent to users.

2.4 Support Vector Machines

2.4.1 Theoretical Background

Support Vector Machines (SVMs) are powerful supervised learning models used for classification and regression. For NLOS detection, we focus on SVM classification.

The core concept of SVM is to find an optimal hyperplane that separates data points of different classes with the maximum margin. The hyperplane is defined by:

$$\mathbf{w} \cdot \mathbf{x} + b = 0 \tag{7}$$

Where:

- $\mathbf{w}$ is the normal vector to the hyperplane
- $\mathbf{x}$ represents feature vectors
- b is the bias term

The decision function for classification is:

$$f(\mathbf{x}) = \text{sign}(\mathbf{w} \cdot \mathbf{x} + b) \tag{8}$$

The margin is defined as the distance between the hyperplane and the closest data points (support vectors). SVM aims to maximize this margin, which leads to the following optimization problem:

$$\min_{\mathbf{w},b} \frac{1}{2}\|\mathbf{w}\|^2 \quad \text{subject to} \quad y_i(\mathbf{w} \cdot \mathbf{x}_i + b) \geq 1 \quad \forall i \tag{9}$$

Where $y_i \in \{-1, 1\}$ are the class labels. This is known as the hard-margin SVM.

For linearly non-separable data, soft-margin SVM introduces slack variables ξ_i to allow for misclassifications:

$$\min_{\mathbf{w},b,\xi} \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^{n} \xi_i \quad \text{subject to} \quad y_i(\mathbf{w} \cdot \mathbf{x}_i + b) \geq 1 - \xi_i, \quad \xi_i \geq 0 \quad \forall i \tag{10}$$

The parameter C controls the trade-off between maximizing the margin and minimizing the training error.

For non-linear classification, SVM uses the "kernel trick" to implicitly map the input features to a higher-dimensional space where the data becomes linearly separable. The kernel function $K(\mathbf{x}_i, \mathbf{x}_j)$ computes the dot product in this higher-dimensional space without explicitly performing the mapping:

$$K(\mathbf{x}_i, \mathbf{x}_j) = \phi(\mathbf{x}_i) \cdot \phi(\mathbf{x}_j) \tag{11}$$

Common kernel functions include:

- Linear: $K(\mathbf{x}_i, \mathbf{x}_j) = \mathbf{x}_i \cdot \mathbf{x}_j$
- Polynomial: $K(\mathbf{x}_i, \mathbf{x}_j) = (\gamma \mathbf{x}_i \cdot \mathbf{x}_j + r)^d$
- Radial Basis Function (RBF): $K(\mathbf{x}_i, \mathbf{x}_j) = e^{-\gamma\|\mathbf{x}_i - \mathbf{x}_j\|^2}$
- Sigmoid: $K(\mathbf{x}_i, \mathbf{x}_j) = \tanh(\gamma \mathbf{x}_i \cdot \mathbf{x}_j + r)$

2.4.2 Application to NLOS Detection

SVM offers several advantages for NLOS detection:

- Effective in high-dimensional spaces, making it suitable for complex signal patterns

- Robust to outliers when using an appropriate C parameter
- Can capture non-linear relationships between features using kernel functions
- Provides probabilistic outputs when calibrated properly
- Works well even with moderate-sized datasets

Linear SVM may be sufficient when the relationship between features (SNR, Elevation, Azimuth) and NLOS status is approximately linear, while kernel-based SVMs (particularly with RBF kernels) can capture more complex, non-linear patterns in the signal characteristics.

The hyperparameter C controls the penalty for misclassification, allowing optimization between false positives (incorrectly identified NLOS signals) and false negatives (missed NLOS signals), which can be critical in GNSS applications where different error types have varying impacts on positioning.

3 Laboratory Exercises

3.1 Exercise 0: Generating Synthetic GPS NLOS Dataset

Objectives

- Understand the characteristics of GPS signal features
- Generate a realistic synthetic dataset for NLOS classification
- Create data representing different environmental scenarios
- Prepare the dataset for subsequent machine learning exercises

Background on GPS Signal Features

To create a realistic synthetic dataset, we need to understand how GPS signal features behave in different environments:

Signal-to-Noise Ratio (SNR)

- LOS signals typically have SNR values in the range of 35-55 dB-Hz
- NLOS signals typically have SNR values in the range of 15-35 dB-Hz
- Very weak signals (SNR $<$ 25 dB-Hz) are often unusable or highly degraded

- SNR tends to correlate positively with elevation angle
- Each constellation has slightly different SNR characteristics

Elevation Angle

- Satellites are typically visible from elevation angles of 5° to 90°
- Satellites below 10-15° elevation are more likely to be blocked by buildings or terrain
- Distribution of elevation angles follows a sinusoidal pattern (more satellites visible at lower elevations)
- NLOS probability decreases approximately exponentially with increasing elevation angle

Azimuth Angle

- Azimuth angles range from 0° to 360°, with uniform distribution in open sky conditions
- In urban environments, buildings create "shadows" that block signals from specific directions
- These blocked directions create patterns in NLOS probability as a function of azimuth

Constellation

- Different constellations (GPS, GLONASS, Galileo, BeiDou) have different orbital properties
- Signal characteristics vary slightly between constellations
- We'll use a simple encoding: 1=GPS, 2=GLONASS, 3=Galileo, 4=BeiDou

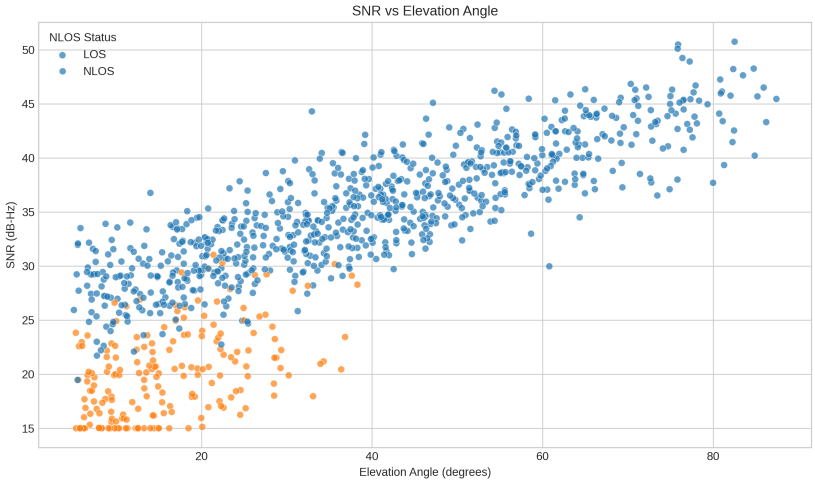

Figure 1: Relationship between SNR and satellite elevation angle. LOS signals (blue) typically have higher SNR values than NLOS signals (red), particularly at higher elevation angles.

Principles of Dataset Generation

The synthetic GPS NLOS dataset is created using several key principles:

1. **Realistic satellite distribution**: Satellite positions follow a sinusoidal distribution in elevation (more satellites at lower elevations) and uniform distribution in azimuth

2. **Environmental modeling**: Three environments are modeled:

   - Urban (500 samples): Dense building environments with high NLOS probability
   - Suburban (300 samples): Moderate building density with medium NLOS probability
   - Open sky (200 samples): Minimal obstructions with low NLOS probability

3. **Building obstruction simulation**: Buildings are modeled as angular sectors defined by:

   - Direction (azimuth angle)
   - Height (maximum elevation angle blocked)
   - Width (angular width in azimuth)

4. **Signal characteristic modeling**: Features are generated with realistic dependencies:
    - SNR increases with elevation and varies by constellation
    - SNR decreases significantly for NLOS signals
    - Random variations added to simulate real-world noise
5. **Probabilistic factors**: Additional random effects simulate:
    - Low-elevation satellites having higher NLOS probability
    - Small buildings/trees causing occasional NLOS conditions
    - Environmental-specific NLOS patterns

This approach creates a dataset that maintains physically realistic relationships between features while incorporating the stochastic nature of real-world signal behavior.

Tasks

1. Import necessary libraries:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from scipy.stats import norm
from mpl_toolkits.mplot3d import Axes3D
```

2. Define functions to generate synthetic data:

```
def generate_sky_distribution(n_satellites=2000, min_elevation=5):
    """
    Generate a realistic distribution of satellite positions in the sky

    Parameters:
    n_satellites (int): Number of satellite observations to generate
    min_elevation (float): Minimum elevation angle in degrees

    Returns:
    tuple: (elevation angles, azimuth angles)
```

```
    """
    # Elevation angles follow a sinusoidal distribution
    # More satellites are visible at lower elevations
    elevation = np.rad2deg(np.arcsin(np.random.uniform(
        np.sin(np.deg2rad(min_elevation)), 1, n_satellites)))

    # Azimuth angles are uniformly distributed around the horizon
    azimuth = np.random.uniform(0, 360, n_satellites)

    return elevation, azimuth

def add_buildings(elevation, azimuth, building_directions,
                  building_heights, building_widths):
    """
    Add simulated buildings that block satellites in specific directions

    Parameters:
    elevation (array): Satellite elevation angles
    azimuth (array): Satellite azimuth angles
    building_directions (list): List of azimuth directions where buildi
    building_heights (list): List of building heights in degrees of elev
    building_widths (list): List of building widths in degrees of azimut

    Returns:
    array: Boolean mask where True indicates the satellite is blocked by
    """
    n_satellites = len(elevation)
    is_blocked = np.zeros(n_satellites, dtype=bool)

    for direction, height, width in zip(building_directions,
                                        building_heights,
                                        building_widths):
        # Calculate angular distance to building direction (considering
        az_distance = np.minimum(
            np.abs(azimuth - direction),
            360 - np.abs(azimuth - direction)
        )

        # Satellite is blocked if it's behind the building and below its
        is_blocked = is_blocked | (
            (az_distance <= width / 2) & (elevation <= height)
```

```
            )

        return is_blocked

    def generate_snr(elevation, constellation, is_nlos):
        """
        Generate realistic SNR values based on elevation, constellation, and
        Parameters:
        elevation (array): Satellite elevation angles
        constellation (array): Satellite constellation indicators (1-4)
        is_nlos (array): Boolean indicator of NLOS status

        Returns:
        array: Simulated SNR values
        """
        n_satellites = len(elevation)

        # Base SNR increases with elevation angle
        base_snr = 25 + (20 * elevation / 90)

        # Add constellation-specific offsets
        constellation_offset = np.zeros(n_satellites)
        constellation_offset[constellation == 1] = 2  # GPS
        constellation_offset[constellation == 2] = 0  # GLONASS
        constellation_offset[constellation == 3] = 3  # Galileo
        constellation_offset[constellation == 4] = 1  # BeiDou

        # Add random variations
        snr = base_snr + constellation_offset + np.random.normal(0, 2, n_sat

        # Reduce SNR for NLOS signals
        nlos_reduction = np.random.uniform(5, 15, n_satellites)
        snr[is_nlos] -= nlos_reduction[is_nlos]

        # Add some noise to make the relationship non-linear
        snr += np.random.normal(0, 2, n_satellites)

        # Ensure SNR stays in a realistic range
        snr = np.clip(snr, 15, 55)
```

```
    return snr

def generate_urban_environment(n_samples=500):
    """
    Generate a dataset simulating an urban environment with buildings

    Parameters:
    n_samples (int): Number of satellite observations to generate

    Returns:
    DataFrame: DataFrame with satellite observations
    """
    # Generate satellite positions
    elevation, azimuth = generate_sky_distribution(n_samples)

    # Randomly assign constellations (1=GPS, 2=GLONASS, 3=Galileo, 4=Bei
    constellation = np.random.choice([1, 2, 3, 4], size=n_samples,
                                     p=[0.4, 0.3, 0.2, 0.1])

    # Define buildings that block signals
    building_directions = [45, 135, 225, 315]  # Buildings in NE, SE, SW
    building_heights = [30, 40, 25, 35]  # Heights in degrees elevation
    building_widths = [60, 40, 50, 45]  # Widths in degrees azimuth

    # Determine NLOS status based on buildings
    is_blocked = add_buildings(elevation, azimuth,
                               building_directions,
                               building_heights,
                               building_widths)

    # Add some randomness to NLOS state (some LOS signals might still be
    # due to smaller buildings, trees, etc. not explicitly modeled)
    for i in range(len(is_blocked)):
        # Low elevation satellites have a chance to be NLOS even if not
        if elevation[i] < 15 and not is_blocked[i]:
            if np.random.random() < 0.3:
                is_blocked[i] = True
        # Even some higher satellites could be NLOS
        elif elevation[i] < 30 and not is_blocked[i]:
            if np.random.random() < 0.1:
                is_blocked[i] = True
```

```
    # Generate SNR values
    snr = generate_snr(elevation, constellation, is_blocked)

    # Create DataFrame
    data = pd.DataFrame({
        'SNR': snr,
        'Constellation': constellation,
        'Elevation': elevation,
        'Azimuth': azimuth,
        'NLOS_Status': is_blocked.astype(int)
    })

    return data

def generate_suburban_environment(n_samples=300):
    """
    Generate a dataset simulating a suburban environment with fewer tall

    Parameters:
    n_samples (int): Number of satellite observations to generate

    Returns:
    DataFrame: DataFrame with satellite observations
    """
    # Generate satellite positions
    elevation, azimuth = generate_sky_distribution(n_samples)

    # Randomly assign constellations (1=GPS, 2=GLONASS, 3=Galileo, 4=Bei
    constellation = np.random.choice([1, 2, 3, 4], size=n_samples,
                                     p=[0.4, 0.3, 0.2, 0.1])

    # Define buildings that block signals (fewer and lower than urban)
    building_directions = [90, 180, 270]  # Buildings in E, S, W
    building_heights = [20, 25, 15]  # Heights in degrees elevation
    building_widths = [30, 25, 35]  # Widths in degrees azimuth

    # Determine NLOS status based on buildings
    is_blocked = add_buildings(elevation, azimuth,
                               building_directions,
                               building_heights,
```

```
                                building_widths)

        # Add some randomness to NLOS state (trees, small buildings, etc.)
        for i in range(len(is_blocked)):
            # Low elevation satellites have a chance to be NLOS even if not
            if elevation[i] < 15 and not is_blocked[i]:
                if np.random.random() < 0.2:
                    is_blocked[i] = True

        # Generate SNR values
        snr = generate_snr(elevation, constellation, is_blocked)

        # Create DataFrame
        data = pd.DataFrame({
            'SNR': snr,
            'Constellation': constellation,
            'Elevation': elevation,
            'Azimuth': azimuth,
            'NLOS_Status': is_blocked.astype(int)
        })

        return data

    def generate_open_sky_environment(n_samples=200):
        """
        Generate a dataset simulating an open sky environment with few obstr

        Parameters:
        n_samples (int): Number of satellite observations to generate

        Returns:
        DataFrame: DataFrame with satellite observations
        """
        # Generate satellite positions
        elevation, azimuth = generate_sky_distribution(n_samples)

        # Randomly assign constellations (1=GPS, 2=GLONASS, 3=Galileo, 4=Bei
        constellation = np.random.choice([1, 2, 3, 4], size=n_samples,
                                         p=[0.4, 0.3, 0.2, 0.1])

        # In open sky, only very low elevation satellites might be NLOS
```

```
    is_blocked = np.zeros(n_samples, dtype=bool)
    for i in range(n_samples):
        if elevation[i] < 10:
            if np.random.random() < 0.15:
                is_blocked[i] = True

    # Generate SNR values
    snr = generate_snr(elevation, constellation, is_blocked)

    # Create DataFrame
    data = pd.DataFrame({
        'SNR': snr,
        'Constellation': constellation,
        'Elevation': elevation,
        'Azimuth': azimuth,
        'NLOS_Status': is_blocked.astype(int)
    })

    return data
```

3. Generate combined dataset from multiple environments:

```
# Set random seed for reproducibility
np.random.seed(42)

# Generate data for different environments
urban_data = generate_urban_environment(500)
suburban_data = generate_suburban_environment(300)
open_sky_data = generate_open_sky_environment(200)

# Combine all datasets
combined_data = pd.concat([urban_data, suburban_data, open_sky_data], ig

# Display basic stats
print("Dataset Shape:", combined_data.shape)
print("\nClass Distribution:")
print(combined_data['NLOS_Status'].value_counts())
print("\nSummary Statistics:")
print(combined_data.describe())
```

```
# Save to CSV file for use in other exercises
combined_data.to_csv('gps_nlos_dataset.csv', index=False)
print("\nDataset saved to 'gps_nlos_dataset.csv'")
```

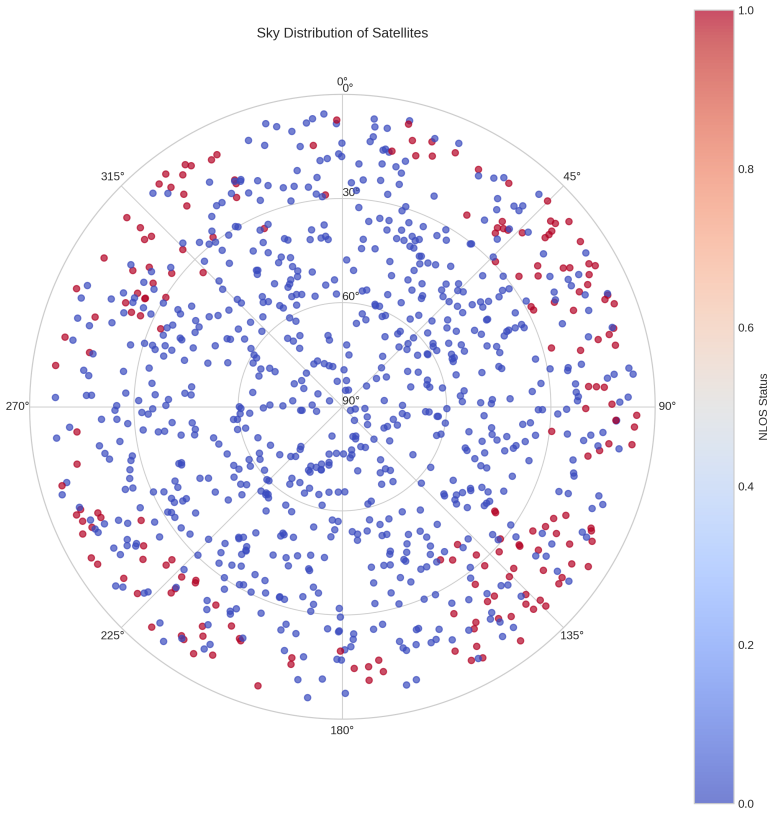

Figure 2: Polar visualization of satellite distribution in the sky. The radial distance corresponds to the complement of elevation angle (90° - elevation), with the center representing zenith (90° elevation) and the outer edge representing the horizon (0° elevation). Colors indicate NLOS status (blue = LOS, red = NLOS).

4. Visualize the dataset:

```
# Plot SNR vs Elevation, colored by NLOS status
plt.figure(figsize=(10, 6))
sns.scatterplot(x='Elevation', y='SNR', hue='NLOS_Status',
                data=combined_data, alpha=0.7)
plt.title('SNR vs Elevation Angle')
```

```
plt.xlabel('Elevation Angle (degrees)')
plt.ylabel('SNR (dB-Hz)')
plt.legend(title='NLOS Status', labels=['LOS', 'NLOS'])
plt.savefig('snr_vs_elevation.png')

# Plot sky distribution using polar coordinates
plt.figure(figsize=(10, 10))
ax = plt.subplot(111, projection='polar')

# Convert azimuth from degrees to radians
azimuth_rad = np.deg2rad(combined_data['Azimuth'])

# In polar plots, 90 degrees - elevation gives the radial distance
radial_distance = 90 - combined_data['Elevation']

# Plot
scatter = ax.scatter(azimuth_rad, radial_distance,
                     c=combined_data['NLOS_Status'], cmap='coolwarm',
                     alpha=0.7, s=30)

# Set up the polar plot
ax.set_theta_zero_location('N')  # 0 degrees at the top
ax.set_theta_direction(-1)  # Clockwise
ax.set_rlabel_position(0)
ax.set_rticks([0, 30, 60, 90])  # From 90 degrees down to 0
ax.set_rlim(0, 90)
ax.set_yticklabels(['90°', '60°', '30°', '0°'])  # Elevation labels

plt.colorbar(scatter, label='NLOS Status')
plt.title('Sky Distribution of Satellites', y=1.08)
plt.savefig('satellite_sky_distribution.png')

# Create 3D visualization of SNR, Elevation, and Azimuth
fig = plt.figure(figsize=(12, 10))
ax = fig.add_subplot(111, projection='3d')

scatter = ax.scatter(combined_data['Azimuth'],
                     combined_data['Elevation'],
                     combined_data['SNR'],
                     c=combined_data['NLOS_Status'],
                     cmap='coolwarm',
```

```
                    s=30, alpha=0.7)

ax.set_xlabel('Azimuth (degrees)')
ax.set_ylabel('Elevation (degrees)')
ax.set_zlabel('SNR (dB-Hz)')
plt.colorbar(scatter, label='NLOS Status')
plt.title('3D Visualization of Signal Features')
plt.savefig('3d_signal_features.png')
```

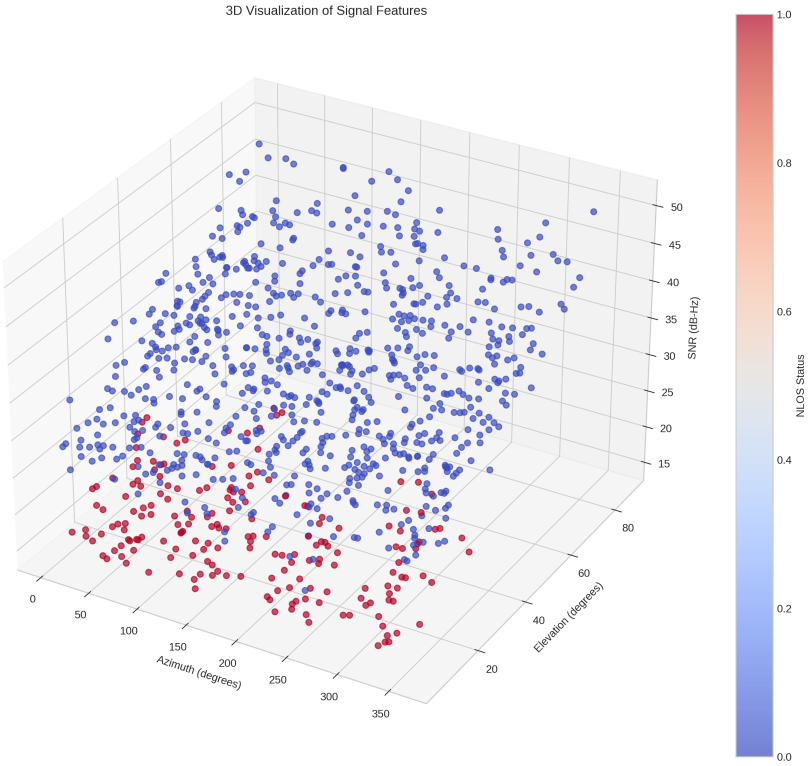

Figure 3: 3D visualization of GPS signal features showing the relationship between Azimuth (x-axis), Elevation (y-axis), and SNR (z-axis). The color represents NLOS status. This visualization helps identify complex patterns that may not be visible in 2D plots.

5. Plot distribution of features by NLOS status:

```
# Plot distribution of features by NLOS status
fig, axs = plt.subplots(2, 2, figsize=(12, 10))
```

```
# SNR distribution
sns.histplot(data=combined_data, x='SNR', hue='NLOS_Status',
             bins=20, ax=axs[0, 0], kde=True)
axs[0, 0].set_title('SNR Distribution by NLOS Status')

# Elevation distribution
sns.histplot(data=combined_data, x='Elevation', hue='NLOS_Status',
             bins=20, ax=axs[0, 1], kde=True)
axs[0, 1].set_title('Elevation Distribution by NLOS Status')

# Azimuth distribution
sns.histplot(data=combined_data, x='Azimuth', hue='NLOS_Status',
             bins=36, ax=axs[1, 0], kde=True)
axs[1, 0].set_title('Azimuth Distribution by NLOS Status')

# Constellation counts
sns.countplot(data=combined_data, x='Constellation', hue='NLOS_Status',
              ax=axs[1, 1])
axs[1, 1].set_title('NLOS Status by Constellation')
axs[1, 1].set_xticks([0, 1, 2, 3])
axs[1, 1].set_xticklabels(['GPS', 'GLONASS', 'Galileo', 'BeiDou'])

plt.tight_layout()
plt.savefig('feature_distributions.png')

# Pairplot for all features
sns.pairplot(combined_data, hue='NLOS_Status', corner=True)
plt.savefig('feature_pairplot.png')
```

6. Display feature correlations:

```
# Calculate correlation matrix
corr_matrix = combined_data.corr()
print("\nCorrelation Matrix:")
print(corr_matrix)

# Plot correlation heatmap
plt.figure(figsize=(10, 8))
sns.heatmap(corr_matrix, annot=True, cmap='coolwarm', fmt='.2f')
```

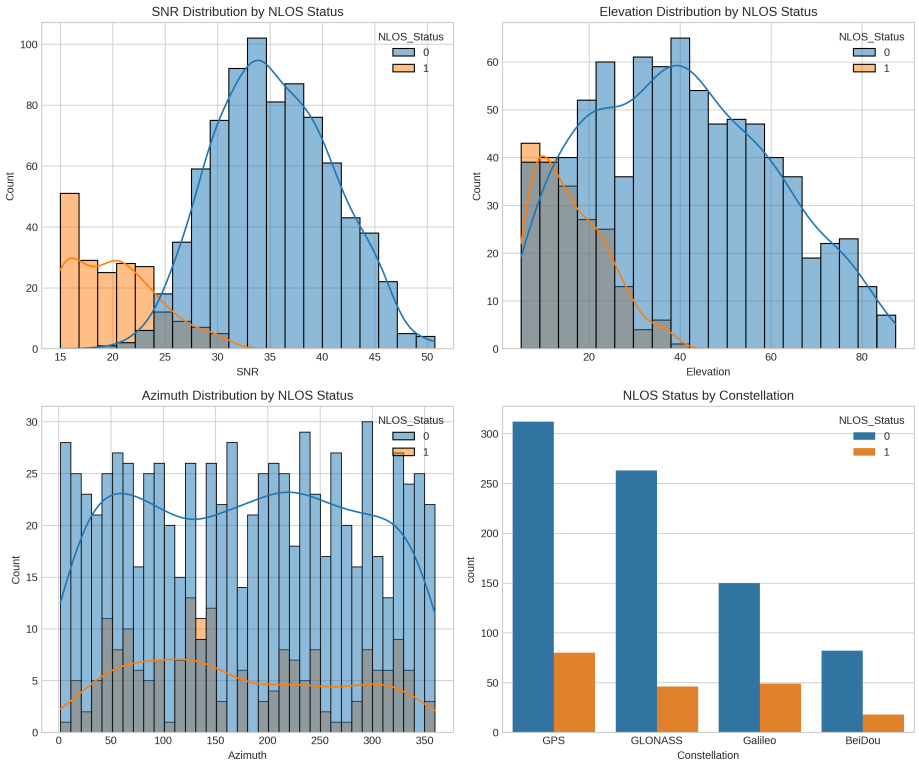

Figure 4: Distributions of key features by NLOS status. Top-left: SNR distribution shows NLOS signals typically have lower values. Top-right: Elevation angle distribution shows NLOS signals are more common at lower elevations. Bottom-left: Azimuth distribution shows some directions have higher NLOS probability. Bottom-right: NLOS probability by satellite constellation.

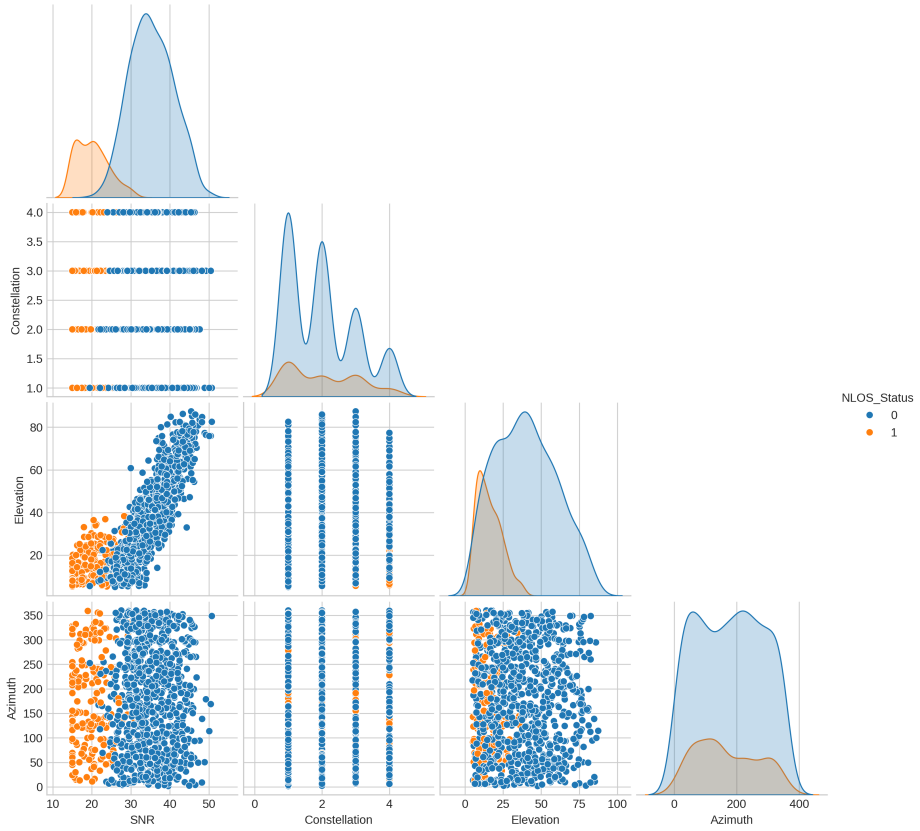

Figure 5: Pairwise relationships between all features, colored by NLOS status. This visualization helps identify potential feature interactions and multivariate patterns that can be exploited by machine learning models.

```
plt.title('Feature Correlation Matrix')
plt.tight_layout()
plt.savefig('correlation_matrix.png')
```

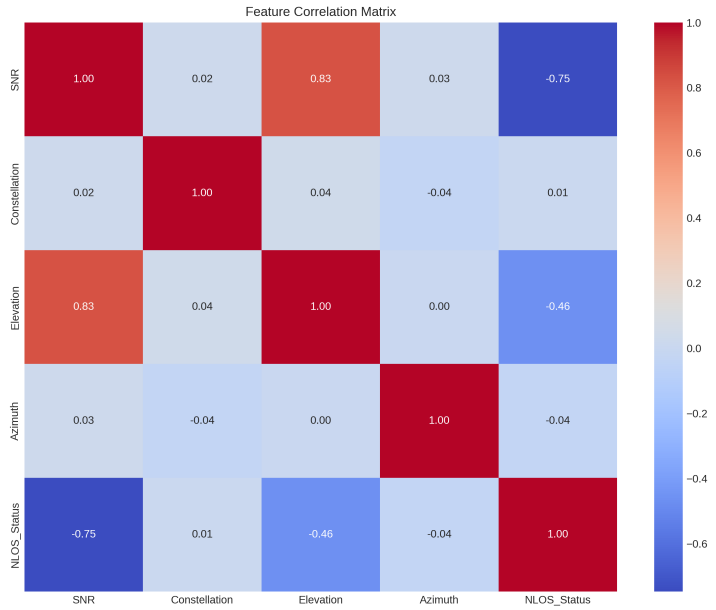

Figure 6: Correlation matrix between features and NLOS status. Strong negative correlations can be observed between NLOS status and both SNR and Elevation angle, confirming that lower SNR and lower elevation angles are associated with higher probability of NLOS signals.

7. Print observations about the dataset:

```
# Print observations about the synthetic dataset
print("\nObservations about the synthetic dataset:")
print(f"1. Total observations: {len(combined_data)}")
print(f"2. NLOS signals: {combined_data['NLOS_Status'].sum()} "
      f"({combined_data['NLOS_Status'].mean()*100:.1f}%)")
print(f"3. Mean SNR for LOS signals: "
      f"{combined_data[combined_data['NLOS_Status']==0]['SNR'].mean():.
print(f"4. Mean SNR for NLOS signals: "
      f"{combined_data[combined_data['NLOS_Status']==1]['SNR'].mean():.
print(f"5. Mean elevation for LOS signals: "
      f"{combined_data[combined_data['NLOS_Status']==0]['Elevation'].mea
```

```
print(f"6. Mean elevation for NLOS signals: "
      f"{combined_data[combined_data['NLOS_Status']==1]['Elevation'].mea
```

3.2 Exercise 1: Data Exploration and Preprocessing

Objectives

- Load and understand the GPS signal dataset
- Visualize the relationships between features and NLOS status
- Prepare data for machine learning models

Dataset Description

The provided dataset contains GPS signal measurements with the following columns:

- `SNR`: Signal-to-Noise Ratio (dB-Hz)
- `Constellation`: Categorical variable indicating satellite system (1=GPS, 2=GLONASS, 3=Galileo, 4=BeiDou)
- `Elevation`: Satellite elevation angle (degrees)
- `Azimuth`: Satellite azimuth angle (degrees)
- `NLOS_Status`: Binary label (0=LOS, 1=NLOS)

Tasks

1. Import necessary libraries:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
```

2. Load the dataset:

```
# Load data
data = pd.read_csv('gps_nlos_dataset.csv')

# Display basic statistics
print(data.describe())
print(data.info())
```

3. Visualize feature distributions by class:

```
# Create pairplot colored by NLOS status
sns.pairplot(data, hue='NLOS_Status')
plt.savefig('feature_relationships.png')

# Create boxplots for each feature by NLOS status
fig, axs = plt.subplots(2, 2, figsize=(12, 10))
sns.boxplot(x='NLOS_Status', y='SNR', data=data, ax=axs[0, 0])
sns.boxplot(x='NLOS_Status', y='Elevation', data=data, ax=axs[0, 1])
sns.boxplot(x='NLOS_Status', y='Azimuth', data=data, ax=axs[1, 0])
sns.countplot(x='Constellation', hue='NLOS_Status', data=data, ax=axs[1
plt.tight_layout()
plt.savefig('feature_boxplots.png')
```

4. Prepare data for modeling:

```
# Split features and target
X = data[['SNR', 'Constellation', 'Elevation', 'Azimuth']]
y = data['NLOS_Status']

# Convert Constellation to one-hot encoding
X = pd.get_dummies(X, columns=['Constellation'], drop_first=True)

# Split into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=42
)

# Scale features (optional for some models)
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
```

```
X_test_scaled = scaler.transform(X_test)
```

3.3 Exercise 2: Implementing Linear Regression

Objectives

- Implement linear regression for NLOS classification
- Evaluate model performance
- Interpret coefficients

Tasks

1. Implement linear regression:

```
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score

# Create and train the model
lin_reg = LinearRegression()
lin_reg.fit(X_train, y_train)

# Make predictions
y_pred_train = lin_reg.predict(X_train)
y_pred_test = lin_reg.predict(X_test)
```

2. Convert continuous predictions to binary classifications:

```
# Convert predictions to binary (0/1)
y_pred_binary = (y_pred_test > 0.5).astype(int)
```

3. Evaluate the model:

```
from sklearn.metrics import accuracy_score, confusion_matrix
from sklearn.metrics import classification_report

# Calculate performance metrics
mse = mean_squared_error(y_test, y_pred_test)
```

```
r2 = r2_score(y_test, y_pred_test)
accuracy = accuracy_score(y_test, y_pred_binary)
conf_matrix = confusion_matrix(y_test, y_pred_binary)

print(f"Mean Squared Error: {mse:.4f}")
print(f"R² Score: {r2:.4f}")
print(f"Accuracy: {accuracy:.4f}")
print("Confusion Matrix:")
print(conf_matrix)
print("\nClassification Report:")
print(classification_report(y_test, y_pred_binary))
```

4. Analyze feature coefficients:

```
# Display coefficients
coefficients = pd.DataFrame({
    'Feature': X.columns,
    'Coefficient': lin_reg.coef_
})
print("Intercept:", lin_reg.intercept_)
print(coefficients.sort_values('Coefficient', ascending=False))

# Visualize coefficients
plt.figure(figsize=(10, 6))
plt.barh(coefficients['Feature'], coefficients['Coefficient'])
plt.xlabel('Coefficient Value')
plt.ylabel('Feature')
plt.title('Linear Regression Coefficients')
plt.tight_layout()
plt.savefig('linear_regression_coefficients.png')
```

5. Interpret the results:

```
# Print interpretation
print("Linear Regression Interpretation:")
print("- Linear regression achieved an accuracy of {:.2f}%".format(accu
print("- The model explains {:.2f}% of the variance in the data".format
print("- Features with the highest influence are:")
for feature, coef in zip(
```

```
        coefficients.nlargest(3, 'Coefficient')['Feature'],
        coefficients.nlargest(3, 'Coefficient')['Coefficient']
    ):
        print(f"  * {feature}: {coef:.4f}")
```

3.4 Exercise 3: Implementing Logistic Regression

Objectives

- Implement logistic regression for NLOS classification
- Evaluate and visualize model performance
- Compare with linear regression

Tasks

1. Implement logistic regression:

```
from sklearn.linear_model import LogisticRegression

# Create and train the model
log_reg = LogisticRegression(max_iter=1000, random_state=42)
log_reg.fit(X_train, y_train)

# Make predictions
y_pred_proba = log_reg.predict_proba(X_test)[:, 1]  # Probability of NL
y_pred = log_reg.predict(X_test)  # Binary prediction
```

2. Evaluate the model:

```
from sklearn.metrics import roc_curve, roc_auc_score

# Calculate performance metrics
accuracy = accuracy_score(y_test, y_pred)
conf_matrix = confusion_matrix(y_test, y_pred)
auc = roc_auc_score(y_test, y_pred_proba)

print(f"Accuracy: {accuracy:.4f}")
print(f"AUC: {auc:.4f}")
```

```
print("Confusion Matrix:")
print(conf_matrix)
print("\nClassification Report:")
print(classification_report(y_test, y_pred))
```

3. Plot ROC curve:

```
# Create ROC curve
fpr, tpr, thresholds = roc_curve(y_test, y_pred_proba)

plt.figure(figsize=(8, 6))
plt.plot(fpr, tpr, label=f'Logistic Regression (AUC = {auc:.3f})')
plt.plot([0, 1], [0, 1], 'k--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve')
plt.legend()
plt.grid(True, alpha=0.3)
plt.savefig('logistic_regression_roc.png')
```

4. Analyze feature coefficients:

```
# Display coefficients
coefficients = pd.DataFrame({
    'Feature': X.columns,
    'Coefficient': log_reg.coef_[0],
    'Odds_Ratio': np.exp(log_reg.coef_[0])
})
print("Intercept:", log_reg.intercept_[0])
print(coefficients.sort_values('Coefficient', ascending=False))

# Visualize coefficients
plt.figure(figsize=(10, 6))
plt.barh(coefficients['Feature'], coefficients['Odds_Ratio'])
plt.xlabel('Odds Ratio (exp(coefficient))')
plt.ylabel('Feature')
plt.title('Logistic Regression Odds Ratios')
plt.axvline(x=1, color='red', linestyle='--')
plt.tight_layout()
```

```
plt.savefig('logistic_regression_odds_ratios.png')
```

5. Create a visualization of the decision boundary (simplified for two features):

```
# For visualization, create a simplified model using the two most import
# Identify the two most important features
top_features = coefficients.abs().sort_values('Coefficient', ascending=F

# Create a model using just these two features
X_train_2d = X_train[top_features]
X_test_2d = X_test[top_features]

log_reg_2d = LogisticRegression(random_state=42)
log_reg_2d.fit(X_train_2d, y_train)

# Create a meshgrid for visualization
x_min, x_max = X_test_2d.iloc[:, 0].min() - 0.5, X_test_2d.iloc[:, 0].ma
y_min, y_max = X_test_2d.iloc[:, 1].min() - 0.5, X_test_2d.iloc[:, 1].ma
xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1),
                     np.arange(y_min, y_max, 0.1))

Z = log_reg_2d.predict(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)

# Plot decision boundary
plt.figure(figsize=(10, 8))
plt.contourf(xx, yy, Z, alpha=0.4)
scatter = plt.scatter(X_test_2d.iloc[:, 0], X_test_2d.iloc[:, 1],
                      c=y_test, edgecolors='k', alpha=0.8)
plt.xlabel(top_features[0])
plt.ylabel(top_features[1])
plt.title('Logistic Regression Decision Boundary')
plt.colorbar(scatter)
plt.savefig('logistic_regression_boundary.png')
```

3.5 Exercise 4: Implementing Decision Trees

Objectives

- Implement a decision tree for NLOS classification
- Visualize the tree structure
- Analyze feature importance

Tasks

1. Implement the decision tree:

```
from sklearn.tree import DecisionTreeClassifier, plot_tree

# Create and train the model
dt_clf = DecisionTreeClassifier(random_state=42, max_depth=5)
dt_clf.fit(X_train, y_train)

# Make predictions
y_pred = dt_clf.predict(X_test)
y_pred_proba = dt_clf.predict_proba(X_test)[:, 1]
```

2. Evaluate the model:

```
# Calculate performance metrics
accuracy = accuracy_score(y_test, y_pred)
conf_matrix = confusion_matrix(y_test, y_pred)
auc = roc_auc_score(y_test, y_pred_proba)

print(f"Accuracy: {accuracy:.4f}")
print(f"AUC: {auc:.4f}")
print("Confusion Matrix:")
print(conf_matrix)
print("\nClassification Report:")
print(classification_report(y_test, y_pred))
```

3. Visualize the decision tree:

```
# Visualize the tree
plt.figure(figsize=(20, 10))
plot_tree(dt_clf, feature_names=X.columns, class_names=['LOS', 'NLOS'],
          filled=True, rounded=True, fontsize=10)
plt.title('Decision Tree for NLOS Detection')
plt.tight_layout()
plt.savefig('decision_tree_visualization.png', dpi=300)
```

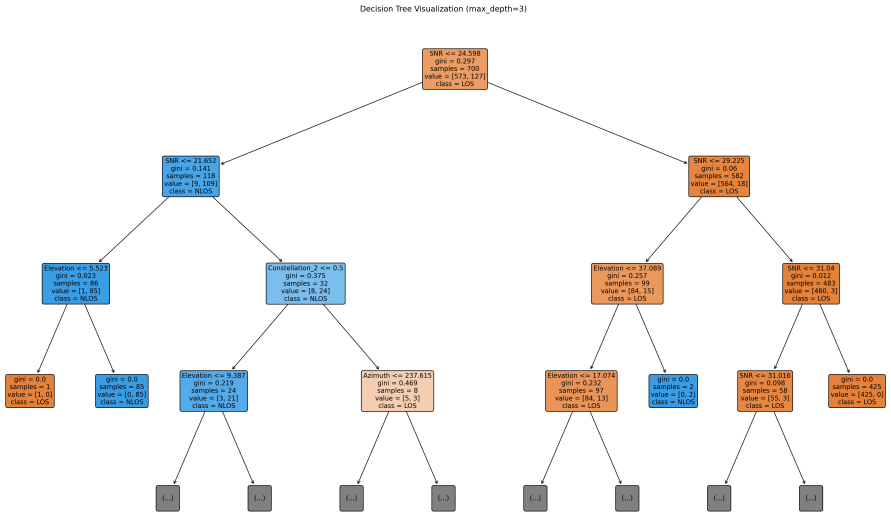

Figure 7: Visualization of the decision tree structure for NLOS detection. Each node shows the decision rule, impurity measure, sample count, and class distribution. Terminal nodes (leaves) show the majority class prediction. The color indicates the majority class (blue for LOS, orange for NLOS) and the intensity represents the class proportion.

4. Create a more detailed tree visualization using graphviz:

```
from sklearn.tree import export_graphviz
import graphviz

# Export the tree as a dot file for graphviz
dot_data = export_graphviz(dt_clf, out_file=None,
                           feature_names=X.columns,
                           class_names=['LOS', 'NLOS'],
                           filled=True, rounded=True,
```

```
                                special_characters=True)

# Render the dot file
graph = graphviz.Source(dot_data)
graph.render("decision_tree_graphviz", format="png")
```

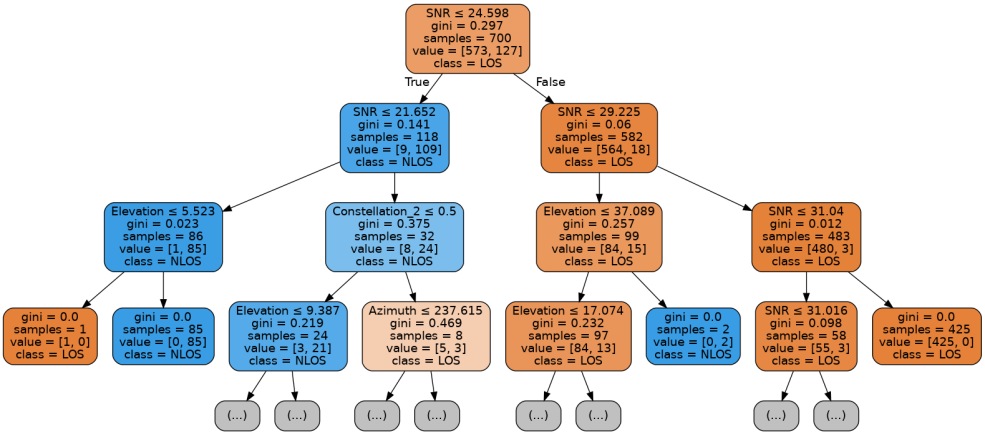

Figure 8: Detailed graphviz visualization of the decision tree for NLOS detection. This visualization shows the complete hierarchical structure of the decision tree with all the splitting rules. The nodes are color-coded by class, with darker shades indicating higher purity, and each split is labeled with the feature and threshold used for the decision.

5. Analyze feature importance:

```
# Display feature importance
importances = pd.DataFrame({
    'Feature': X.columns,
    'Importance': dt_clf.feature_importances_
}).sort_values('Importance', ascending=False)

print("Feature Importance:")
print(importances)

# Visualize feature importance
plt.figure(figsize=(10, 6))
plt.barh(importances['Feature'], importances['Importance'])
plt.xlabel('Importance')
```

```
plt.ylabel('Feature')
plt.title('Decision Tree Feature Importance')
plt.tight_layout()
plt.savefig('decision_tree_importance.png')
```

6. Experiment with tree depth:

```
# Test different tree depths
depths = range(1, 21)
train_accuracy = []
test_accuracy = []

for depth in depths:
    dt = DecisionTreeClassifier(max_depth=depth, random_state=42)
    dt.fit(X_train, y_train)

    train_accuracy.append(accuracy_score(y_train, dt.predict(X_train)))
    test_accuracy.append(accuracy_score(y_test, dt.predict(X_test)))

# Plot accuracy vs tree depth
plt.figure(figsize=(10, 6))
plt.plot(depths, train_accuracy, marker='o', label='Training Accuracy')
plt.plot(depths, test_accuracy, marker='o', label='Testing Accuracy')
plt.xlabel('Maximum Tree Depth')
plt.ylabel('Accuracy')
plt.title('Accuracy vs Tree Depth')
plt.legend()
plt.grid(True, alpha=0.3)
plt.savefig('decision_tree_depth_analysis.png')
```

7. Compare ROC curves for all models:

```
# Re-run all models to collect predictions
# (Assuming you've already created the models in previous exercises)

# Make predictions for all models
lr_pred_proba = lin_reg.predict(X_test)
lg_pred_proba = log_reg.predict_proba(X_test)[:, 1]
dt_pred_proba = dt_clf.predict_proba(X_test)[:, 1]
```

```
# Calculate ROC curves
lr_fpr, lr_tpr, _ = roc_curve(y_test, lr_pred_proba)
lg_fpr, lg_tpr, _ = roc_curve(y_test, lg_pred_proba)
dt_fpr, dt_tpr, _ = roc_curve(y_test, dt_pred_proba)

# Calculate AUC scores
lr_auc = roc_auc_score(y_test, lr_pred_proba)
lg_auc = roc_auc_score(y_test, lg_pred_proba)
dt_auc = roc_auc_score(y_test, dt_pred_proba)

# Plot all ROC curves
plt.figure(figsize=(10, 8))
plt.plot(lr_fpr, lr_tpr, label=f'Linear Regression (AUC = {lr_auc:.3f})
plt.plot(lg_fpr, lg_tpr, label=f'Logistic Regression (AUC = {lg_auc:.3f
plt.plot(dt_fpr, dt_tpr, label=f'Decision Tree (AUC = {dt_auc:.3f})')
plt.plot([0, 1], [0, 1], 'k--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curves Comparison')
plt.legend(loc='lower right')
plt.grid(True, alpha=0.3)
plt.savefig('model_comparison_roc.png')
```

3.6 Exercise 5: Implementing Support Vector Machines

Objectives

- Implement an SVM classifier for NLOS detection
- Explore different kernel functions
- Evaluate performance and tune hyperparameters

Tasks

1. Implement linear SVM:

```
from sklearn.svm import SVC

# Create and train a linear SVM model
```

```
svm_linear = SVC(kernel='linear', probability=True, random_state=42)
svm_linear.fit(X_train, y_train)

# Make predictions
y_pred_linear = svm_linear.predict(X_test)
y_pred_proba_linear = svm_linear.predict_proba(X_test)[:, 1]
```

2. Implement SVM with RBF kernel:

```
# Create and train an SVM model with RBF kernel
svm_rbf = SVC(kernel='rbf', probability=True, random_state=42)
svm_rbf.fit(X_train, y_train)

# Make predictions
y_pred_rbf = svm_rbf.predict(X_test)
y_pred_proba_rbf = svm_rbf.predict_proba(X_test)[:, 1]
```

3. Evaluate the models:

```
# Calculate performance metrics for linear SVM
accuracy_linear = accuracy_score(y_test, y_pred_linear)
conf_matrix_linear = confusion_matrix(y_test, y_pred_linear)
auc_linear = roc_auc_score(y_test, y_pred_proba_linear)

print("Linear SVM Results:")
print(f"Accuracy: {accuracy_linear:.4f}")
print(f"AUC: {auc_linear:.4f}")
print("Confusion Matrix:")
print(conf_matrix_linear)
print("\nClassification Report:")
print(classification_report(y_test, y_pred_linear))

# Calculate performance metrics for RBF SVM
accuracy_rbf = accuracy_score(y_test, y_pred_rbf)
conf_matrix_rbf = confusion_matrix(y_test, y_pred_rbf)
auc_rbf = roc_auc_score(y_test, y_pred_proba_rbf)

print("\nRBF SVM Results:")
print(f"Accuracy: {accuracy_rbf:.4f}")
```

```
print(f"AUC: {auc_rbf:.4f}")
print("Confusion Matrix:")
print(conf_matrix_rbf)
print("\nClassification Report:")
print(classification_report(y_test, y_pred_rbf))
```

4. Visualize decision boundaries for linear SVM (simplified for two features):

```
# Identify the two most important features (using coefficients from line
coefficients = np.abs(svm_linear.coef_[0])
feature_indices = np.argsort(coefficients)[-2:]
top_features = [X.columns[i] for i in feature_indices]

# Create a model using just these two features
X_train_2d = X_train[top_features]
X_test_2d = X_test[top_features]

svm_2d = SVC(kernel='linear', probability=True, random_state=42)
svm_2d.fit(X_train_2d, y_train)

# Create a meshgrid for visualization
x_min, x_max = X_test_2d.iloc[:, 0].min() - 0.5, X_test_2d.iloc[:, 0].ma
y_min, y_max = X_test_2d.iloc[:, 1].min() - 0.5, X_test_2d.iloc[:, 1].ma
xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1),
                     np.arange(y_min, y_max, 0.1))

Z = svm_2d.predict(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)

# Plot decision boundary
plt.figure(figsize=(10, 8))
plt.contourf(xx, yy, Z, alpha=0.4)
scatter = plt.scatter(X_test_2d.iloc[:, 0], X_test_2d.iloc[:, 1],
                      c=y_test, edgecolors='k', alpha=0.8)
plt.xlabel(top_features[0])
plt.ylabel(top_features[1])
plt.title('Linear SVM Decision Boundary')
plt.colorbar(scatter)
plt.savefig('svm_linear_boundary.png')
```

5. Tune SVM hyperparameters:

```
from sklearn.model_selection import GridSearchCV

# Define parameter grid for RBF kernel
param_grid = {
    'C': [0.1, 1, 10, 100],
    'gamma': ['scale', 'auto', 0.1, 0.01, 0.001]
}

# Create grid search object
grid_search = GridSearchCV(
    SVC(kernel='rbf', probability=True, random_state=42),
    param_grid=param_grid,
    cv=5,
    scoring='roc_auc',
    verbose=1,
    n_jobs=-1
)

# Fit grid search
grid_search.fit(X_train, y_train)

# Print best parameters
print("Best Parameters:", grid_search.best_params_)
print("Best Cross-Validation Score:", grid_search.best_score_)

# Create and evaluate best model
best_svm = grid_search.best_estimator_
y_pred_best = best_svm.predict(X_test)
y_pred_proba_best = best_svm.predict_proba(X_test)[:, 1]

accuracy_best = accuracy_score(y_test, y_pred_best)
auc_best = roc_auc_score(y_test, y_pred_proba_best)

print(f"Optimized SVM - Accuracy: {accuracy_best:.4f}")
print(f"Optimized SVM - AUC: {auc_best:.4f}")
print("Classification Report:")
print(classification_report(y_test, y_pred_best))
```

6. Compare ROC curves for different SVM models:

```
# Calculate ROC curves
linear_fpr, linear_tpr, _ = roc_curve(y_test, y_pred_proba_linear)
rbf_fpr, rbf_tpr, _ = roc_curve(y_test, y_pred_proba_rbf)
best_fpr, best_tpr, _ = roc_curve(y_test, y_pred_proba_best)

# Plot all ROC curves
plt.figure(figsize=(10, 8))
plt.plot(linear_fpr, linear_tpr, label=f'Linear SVM (AUC = {auc_linear:
plt.plot(rbf_fpr, rbf_tpr, label=f'RBF SVM (AUC = {auc_rbf:.3f})')
plt.plot(best_fpr, best_tpr, label=f'Optimized SVM (AUC = {auc_best:.3f
plt.plot([0, 1], [0, 1], 'k--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('SVM ROC Curves Comparison')
plt.legend(loc='lower right')
plt.grid(True, alpha=0.3)
plt.savefig('svm_roc_comparison.png')
```

3.7 Exercise 6: Model Comparison and Real-World Application

Objectives

- Compare the performance of all implemented models
- Apply the best model to a realistic scenario
- Discuss practical implications of NLOS detection

Tasks

1. Create a performance comparison table:

```
# Create comparison dataframe
models = ['Linear Regression', 'Logistic Regression',
          'Decision Tree', 'SVM (Linear)', 'SVM (RBF)']
```

```
accuracy_scores = [
    accuracy_score(y_test, (lin_reg.predict(X_test) > 0.5).astype(int)),
    accuracy_score(y_test, log_reg.predict(X_test)),
    accuracy_score(y_test, dt_clf.predict(X_test)),
    accuracy_score(y_test, svm_linear.predict(X_test)),
    accuracy_score(y_test, best_svm.predict(X_test))
]

auc_scores = [
    roc_auc_score(y_test, lin_reg.predict(X_test)),
    roc_auc_score(y_test, log_reg.predict_proba(X_test)[:, 1]),
    roc_auc_score(y_test, dt_clf.predict_proba(X_test)[:, 1]),
    roc_auc_score(y_test, svm_linear.predict_proba(X_test)[:, 1]),
    roc_auc_score(y_test, best_svm.predict_proba(X_test)[:, 1])
]

comparison_df = pd.DataFrame({
    'Model': models,
    'Accuracy': accuracy_scores,
    'AUC': auc_scores
})

print("Model Performance Comparison:")
print(comparison_df.sort_values('AUC', ascending=False))

# Create a bar chart comparison
plt.figure(figsize=(12, 6))

x = np.arange(len(models))
width = 0.35

plt.bar(x - width/2, accuracy_scores, width, label='Accuracy')
plt.bar(x + width/2, auc_scores, width, label='AUC')

plt.xlabel('Model')
plt.ylabel('Score')
plt.title('Model Performance Comparison')
plt.xticks(x, models, rotation=45)
plt.legend()
plt.tight_layout()
plt.savefig('model_comparison.png')
```

2. Apply the best model to a realistic scenario:

```
# Assuming SVM with RBF kernel is the best model
# Let's create a simulated walking path in an urban environment

def generate_urban_path_data(n_points=100):
    """Generate synthetic GPS data for a path through an urban area"""
    # Time in seconds
    time = np.arange(n_points)

    # Generate synthetic features
    snr = np.random.normal(35, 10, n_points)  # Average SNR with noise
    constellation = np.random.choice([1, 2, 3, 4], n_points)  # Random c

    # Create a pattern for elevation (satellites movement)
    elevation = 45 + 15 * np.sin(time / 30) + np.random.normal(0, 5, n_p
    elevation = np.clip(elevation, 5, 90)

    # Create azimuth angles that change as the user moves
    azimuth = (time * 3) % 360 + np.random.normal(0, 5, n_points)

    # Create buildings at certain azimuths to cause NLOS
    building_azimuths = [30, 150, 270]
    building_widths = [20, 30, 15]

    # Start with all LOS
    nlos_status = np.zeros(n_points)

    # Add NLOS based on buildings and low elevation
    for b_azimuth, b_width in zip(building_azimuths, building_widths):
        for i in range(n_points):
            # If azimuth is pointing toward a building and elevation is
            if (abs((azimuth[i] - b_azimuth) % 360) < b_width and
                elevation[i] < 30):
                nlos_status[i] = 1
            # Low elevation often means NLOS
            elif elevation[i] < 15:
                nlos_status[i] = 1
            # Very low SNR is likely NLOS
```

```
            elif snr[i] < 20:
                nlos_status[i] = 1

    # Create dataframe
    path_df = pd.DataFrame({
        'Time': time,
        'SNR': snr,
        'Constellation': constellation,
        'Elevation': elevation,
        'Azimuth': azimuth,
        'NLOS_Status': nlos_status
    })

    return path_df

# Generate and display synthetic path data
path_data = generate_urban_path_data(200)
print(path_data.head())

# One-hot encode Constellation
path_features = pd.get_dummies(
    path_data[['SNR', 'Constellation', 'Elevation', 'Azimuth']],
    columns=['Constellation'],
    drop_first=True
)

# Add any missing columns that the model was trained on
for col in X.columns:
    if col not in path_features.columns:
        path_features[col] = 0

# Ensure columns are in the same order as during training
path_features = path_features[X.columns]

# Apply the SVM model to detect NLOS
path_data['NLOS_Predicted'] = best_svm.predict(path_features)
path_data['NLOS_Probability'] = best_svm.predict_proba(path_features)[:

# Calculate accuracy of the model on this synthetic data
accuracy = accuracy_score(path_data['NLOS_Status'], path_data['NLOS_Pred
print(f"Accuracy on synthetic urban path: {accuracy:.4f}")
```

```
# Visualize the path and NLOS predictions
plt.figure(figsize=(15, 10))

# Plot 1: Path with NLOS status
plt.subplot(2, 1, 1)
plt.scatter(path_data['Time'], path_data['Elevation'],
           c=path_data['NLOS_Status'], cmap='coolwarm',
           alpha=0.8, s=50)
plt.colorbar(label='Actual NLOS Status')
plt.xlabel('Time (seconds)')
plt.ylabel('Elevation Angle (degrees)')
plt.title('Actual NLOS Status Along Path')

# Plot 2: Path with NLOS predictions
plt.subplot(2, 1, 2)
plt.scatter(path_data['Time'], path_data['Elevation'],
           c=path_data['NLOS_Probability'], cmap='coolwarm',
           alpha=0.8, s=50)
plt.colorbar(label='Predicted NLOS Probability')
plt.xlabel('Time (seconds)')
plt.ylabel('Elevation Angle (degrees)')
plt.title('Predicted NLOS Probability Along Path')

plt.tight_layout()
plt.savefig('urban_path_nlos_prediction.png')

# Plot Azimuth visualization (polar plot)
plt.figure(figsize=(10, 10))
ax = plt.subplot(111, projection='polar')

# Convert azimuth from degrees to radians
azimuth_rad = np.deg2rad(path_data['Azimuth'])

# Create scatter plot
sc = ax.scatter(azimuth_rad, path_data['Elevation'],
               c=path_data['NLOS_Probability'], cmap='coolwarm',
               alpha=0.8, s=50)

# Set plot properties
ax.set_theta_zero_location('N')  # Set 0 degrees to North
```

```
ax.set_theta_direction(-1)  # Clockwise
ax.set_rlabel_position(0)  # Move radial labels away from plot

# Set custom radial limits and labels
ax.set_rmax(90)
ax.set_rticks([0, 30, 60, 90])
ax.set_rlabel_position(0)

plt.colorbar(sc, label='NLOS Probability')
plt.title('NLOS Probability by Azimuth and Elevation', y=1.08)
plt.tight_layout()
plt.savefig('nlos_skyplot.png')
```

3. Discussion and practical application:

```
# Create a function to simulate how NLOS detection improves positioning
def simulate_positioning_improvement(path_data):
    """Simulate positioning improvement with NLOS detection"""
    # Create synthetic positioning errors
    # LOS signals: small errors, NLOS signals: large errors
    np.random.seed(42)
    path_data['Position_Error_Without_NLOS_Detection'] = np.where(
        path_data['NLOS_Status'] == 1,
        np.random.normal(15, 5, len(path_data)),  # Large errors for NL
        np.random.normal(2, 1, len(path_data))    # Small errors for LOS
    )

    # With NLOS detection: exclude high probability NLOS signals
    path_data['Used_For_Position'] = path_data['NLOS_Probability'] < 0.7

    # Calculate position error with NLOS detection
    # If we have enough satellites after filtering (at least 4)
    min_satellites = 4
    path_data['Available_Satellites'] = 10  # Assume 10 satellites visib

    position_errors = []

    for i in range(len(path_data)):
        # Count satellites used after NLOS filtering
        satellites_used = path_data['Available_Satellites'][i] * sum(
```

```
            path_data['Used_For_Position'][max(0, i-9):i+1]
        ) / min(10, i+1)

        if satellites_used >= min_satellites:
            # Good position fix with only LOS satellites
            error = np.random.normal(2, 1)
        else:
            # Have to use some NLOS satellites
            error = np.random.normal(5, 3)

        position_errors.append(error)

    path_data['Position_Error_With_NLOS_Detection'] = position_errors

    return path_data

# Apply the simulation
path_data = simulate_positioning_improvement(path_data)

# Plot positioning error comparison
plt.figure(figsize=(12, 6))
plt.plot(path_data['Time'], path_data['Position_Error_Without_NLOS_Detec
         'r-', label='Without NLOS Detection')
plt.plot(path_data['Time'], path_data['Position_Error_With_NLOS_Detectio
         'g-', label='With NLOS Detection')
plt.xlabel('Time (seconds)')
plt.ylabel('Position Error (meters)')
plt.title('Impact of NLOS Detection on Positioning Accuracy')
plt.legend()
plt.grid(True, alpha=0.3)
plt.savefig('positioning_improvement.png')

# Calculate and print summary statistics
avg_error_without = path_data['Position_Error_Without_NLOS_Detection'].m
avg_error_with = path_data['Position_Error_With_NLOS_Detection'].mean()
improvement = (1 - avg_error_with / avg_error_without) * 100

print(f"Average position error without NLOS detection: {avg_error_withou
print(f"Average position error with NLOS detection: {avg_error_with:.2f}
print(f"Positioning accuracy improvement: {improvement:.1f}%")
```

4 Discussion Questions

After completing the laboratory exercises, consider and discuss the following questions:

1. How do the different machine learning methods compare in terms of accuracy, interpretability, and computational efficiency for NLOS detection?

2. Which features proved most important for identifying NLOS signals? How do these align with theoretical expectations?

3. How might NLOS detection techniques be integrated into real-time positioning systems?

4. What other features not included in this lab might improve NLOS detection performance?

5. How might the performance of these algorithms vary in different environments (urban canyons, open areas, indoor spaces)?

6. How could transfer learning be applied to adapt a model trained in one environment to perform well in a different environment?

7. What are the limitations of machine learning approaches for NLOS detection compared to other methods (e.g., 3D mapping, antenna array processing)?

8. How could you extend the models to not just identify NLOS signals but also correct for their effects?

5 Additional Resources

- Groves, P.D. (2013). Principles of GNSS, inertial, and multisensor integrated navigation systems. Artech House.

- Hsu, L.T. (2018). GNSS multipath detection using a machine learning approach. IEEE International Conference on Intelligent Transportation Systems (ITSC), pp. 3088-3093.

- James, G., Witten, D., Hastie, T., & Tibshirani, R. (2013). An introduction to statistical learning. Springer.

- Cortes, C., Vapnik, V. (1995). Support-vector networks. Machine learning, 20(3), 273-297.
- scikit-learn documentation: `https://scikit-learn.org/stable/`