BE 521: Homework 5 Questions

Spring 2025

66 points

Due: Feb 27th, 2025

Objective: HFO detection and cross-validation

AI Usage Notice

The use of artificial intelligence tools (e.g., large language models, code assistants) is permitted. However, students must explicitly state the specific ways AI was used in completing their work. Failure to disclose AI usage may result in an oral examination to assess understanding, at the discretion of Dr. Litt.

If AI was used in the completion of this assignment, please provide a statement below:

[Enter your statement here]

HFO Dataset

High frequency oscillations (HFOs) are quasi-periodic intracranial EEG transients with durations on the order of tens of milliseconds and peak frequencies in the range of 80 to 500 Hz. There has been considerable interest among the epilepsy research community in the potential of these signals as biomarkers for epileptogenic networks.

In this homework exercise, you will explore a dataset of candidate HFOs detected using the algorithm of Staba et al. (see article on Canvas). The raw recordings from which this dataset arises come from a human subject with mesial temporal lobe epilepsy and were contributed by the laboratory of Dr. Greg Worrell at the Mayo Clinic in Rochester, MN.

The dataset `I521_A0004_D001` contains raw HFO clips that are normalized to zero mean and unit standard deviation but are otherwise unprocessed. The raw dataset contains two channels of data: Test_raw_norm and Train_raw_norm, storing raw testing and training sets of HFO clips respectively. The raw dataset also contains two annotation layers: Testing windows and Training windows, storing HFO clip start and stop times (in microseconds) for each of the two channels above. Annotations contain the classification by an "expert" reviewer (i.e., a doctor) of each candidate HFO as either an HFO (2) or an artifact (1). On ieeg.org and upon downloading the annotations, you can view this in the "description" field.

In [76]:
```
#Set up the notebook environment
!pip install git+https://github.com/ieeg-portal/ieegpy.git # Install ieegpy toolbox directly from github
from ieeg.auth import Session
from IPython.display import Image
import matplotlib
import matplotlib.pyplot as plt
import numpy as np
from scipy import signal as sig
import datetime
import time
import sklearn
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
from sklearn.neighbors import KNeighborsClassifier
```

```
Collecting git+https://github.com/ieeg-portal/ieegpy.git
  Cloning https://github.com/ieeg-portal/ieegpy.git to /tmp/pip-req-build-b3ysd85t
  Running command git clone --filter=blob:none --quiet https://github.com/ieeg-portal/ieegpy.git /tmp/pip-req-bu
ild-b3ysd85t
  Resolved https://github.com/ieeg-portal/ieegpy.git to commit 080bfa42a8503380ef164b5e7b116613f75073bb
  Preparing metadata (setup.py) ... done
Requirement already satisfied: deprecation in /usr/local/lib/python3.11/dist-packages (from ieeg==1.6) (2.1.0)
Requirement already satisfied: requests in /usr/local/lib/python3.11/dist-packages (from ieeg==1.6) (2.32.3)
Requirement already satisfied: numpy in /usr/local/lib/python3.11/dist-packages (from ieeg==1.6) (1.26.4)
Requirement already satisfied: pandas in /usr/local/lib/python3.11/dist-packages (from ieeg==1.6) (2.2.2)
Requirement already satisfied: pennprov==2.2.4 in /usr/local/lib/python3.11/dist-packages (from ieeg==1.6) (2.2.
4)
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.11/dist-packages (from pennprov==2.2
.4->ieeg==1.6) (2025.1.31)
Requirement already satisfied: python-dateutil>=2.1 in /usr/local/lib/python3.11/dist-packages (from pennprov==2
.2.4->ieeg==1.6) (2.8.2)
Requirement already satisfied: six>=1.10 in /usr/local/lib/python3.11/dist-packages (from pennprov==2.2.4->ieeg=
=1.6) (1.17.0)
Requirement already satisfied: urllib3>=1.23 in /usr/local/lib/python3.11/dist-packages (from pennprov==2.2.4->i
eeg==1.6) (2.3.0)
Requirement already satisfied: packaging in /usr/local/lib/python3.11/dist-packages (from deprecation->ieeg==1.6
) (24.2)
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.11/dist-packages (from pandas->ieeg==1.6)
(2025.1)
Requirement already satisfied: tzdata>=2022.7 in /usr/local/lib/python3.11/dist-packages (from pandas->ieeg==1.6
) (2025.1)
Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/python3.11/dist-packages (from request
s->ieeg==1.6) (3.4.1)
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.11/dist-packages (from requests->ieeg==1.6
) (3.10)
```

Let's familiarize ourselves with IEEG's annotation object.

1. From an IEEG `Dataset` object, we can call `.get_annotation_layers()`, which returns a dictionary where the keys are the names of the annotation layers, and the associated values are the number of annotations per layer.
2. We can load annotations from IEEG.org using the `.get_annotations(ANNOTATION_LAYER_NAME)` function. This returns a list of IEEG `Annotation` objects for the annotation layer you specified (by name).
3. The `Annotation` object itself has three important attributes:
    - `.description` tells you the value of the annotation (1 for Artifact, 2 for HFO)
    - `.start_time_offset_usec` tells you the starting timepoint of the annotation in microseconds
    - `.end_time_offset_usec` tells you the ending timepoint of the annotation in microseconds.

We give you a `load_annotations(dataset, annotation_layer_name)` function to easily load annotations from IEEG. This function takes two arguments: the IEEG `Dataset` object and the name of the annotation layer you want to load (given to you using `.get_annotation_layers()`). The function returns two objects. The first is an (n x 3) array with each of the n annotation's description, start_time_offset_usec, and end_time_offset_usec, respectively. The second is a list with the `Annotation` objects themselves.

In [77]:
```python
def load_annotations(dataset, annotation_layer_name):
  """
  Loads annotations for the specified annotation layer
  Input: 2 arguments
    dataset: the IEEG Dataset object
    annotation_layer_name: the name of the annotation layer (str)
  Returns: 2 objects
    (n x 3) numpy array. [i, :] contains the description, start_time_offset_usec, and end_time_offset_usec of a
    n-sized list. The raw output of the IEEG .get_annotations() function
  """
  #load the annotations
  annotations = dataset.get_annotations(annotation_layer_name)

  #container for annotation information. Set all default values to -1.
  anno_info = np.zeros((len(annotations), 3)) - 1

  #for each annotation, get the description, start and end times
  for i in range(len(annotations)):
    anno_info[i, 0] = annotations[i].description
    anno_info[i, 1] = annotations[i].start_time_offset_usec
    anno_info[i, 2] = annotations[i].end_time_offset_usec

  return [anno_info, annotations]
```

For convenience, we'll also give you `load_full_channels` again.

In [78]:
```python
def load_full_channels(dataset, duration_secs, sampling_rate, chn_idx):
```

```
  """
  Loads the entire channel from IEEG.org
  Input:
    dataset: the IEEG dataset object
    duration_secs: the duration of the channel, in seconds
    sampling_rate: the sampling rate of the channel, in Hz
    chn_idx: the indicies of the m channels you want to load, as an array-like object
  Returns:
    [n, m] numpy array of the channels' values.
  """
  #stores the segments of the channel's data
  chn_segments = []

  #how many segments do we expect?
  num_segments = int(np.ceil(duration_secs * sampling_rate / 6e5))

  #segment start times and the step
  seg_start, step = np.linspace(1, duration_secs*1e6, num_segments, endpoint=False, retstep=True)

  #get the segments
  for start in seg_start:
    chn_segments.append(dataset.get_data(start, step, chn_idx))

  #concatenate the segments vertically
  return np.vstack(chn_segments)
```

1. Simulating the SABA Detector (12 pts)

Candidate HFO clips were detected with the Staba et al. algorithm and subsequently validated by an expert as a true HFO or not. In this first section, we will use the original iEEG clips containing HFOs and re-simulate a portion of the Staba detection.

1

How many samples exist for each class (HFO vs. artifact) in the training set? (Show code to support your answer.) (1 pts)

In [79]:
```
#your code here
with open('/content/pra_ieeglogin(9).bin', 'r') as f:
  session = Session('prasadpr', f.read())
dataset = session.open_dataset('I521_A0004_D001')

a = load_annotations(dataset, 'Training windows')

num_artifacts = sum(1 for x in a[0] if x[0] == 1)
num_hfos = sum(1 for x in a[0] if x[0] == 2)

print(f"Number of artifacts: {num_artifacts}")
print(f"Number of HFOs: {num_hfos}")
```

```
Number of artifacts: 99
Number of HFOs: 101
```

Your answer here

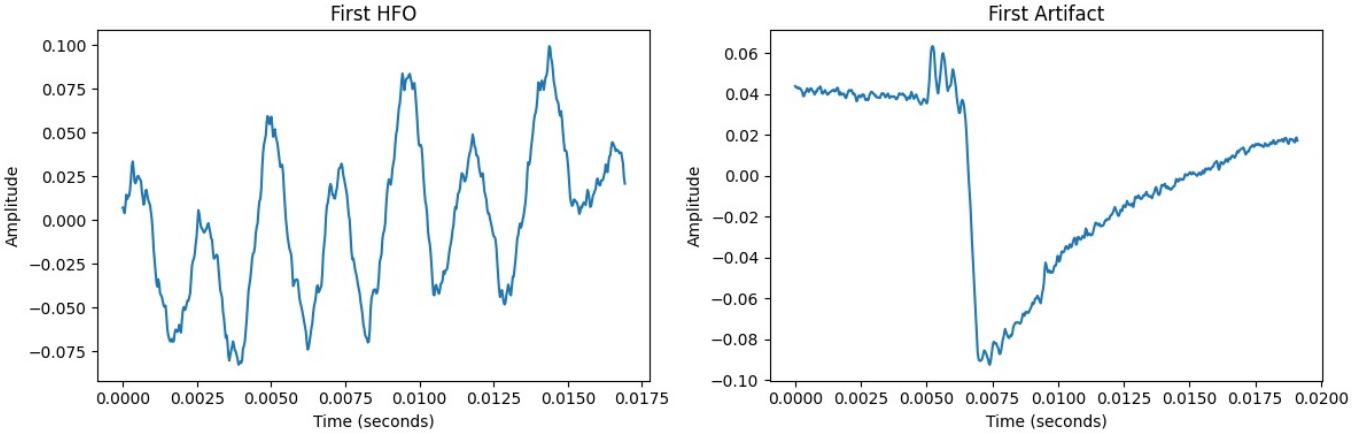
2

Using the training set, find the first occurrence of the first valid HFO and the first artifact. Using `plt.subplots` with 2 plots, plot the HFO on the left and artifact on the right. (2 pts)

In [80]:
```
import numpy as np
import matplotlib.pyplot as plt

# Assuming you have already set up the IEEG session and loaded the dataset
with open('/content/pra_ieeglogin(9).bin', 'r') as f:
    session = Session('prasadpr', f.read())
dataset = session.open_dataset('I521_A0004_D001')

# Check available channels in the dataset
channels = dataset.get_channel_labels()
print("Available channels:", channels)

# Load annotations for the "Training windows" layer
annotations_array, annotations_list = load_annotations(dataset, 'Training windows')

# Find the first HFO (description = 2) and the first artifact (description = 1)
first_hfo = None
first_artifact = None

for annotation in annotations_array:
    if annotation[0] == 2 and first_hfo is None:  # First HFO
```

```
        first_hfo = annotation
    if annotation[0] == 1 and first_artifact is None:  # First artifact
        first_artifact = annotation
    if first_hfo is not None and first_artifact is not None:
        break

# Function to extract raw data using load_full_channels
def extract_raw_data(dataset, start_time_usec, end_time_usec, sampling_rate, channel_name):
    # Convert microseconds to seconds
    start_time_sec = start_time_usec
    end_time_sec = end_time_usec # in microsec
    duration_secs = end_time_sec - start_time_sec

    # Get the index of the channel
    channel_index = dataset.get_channel_indices([channel_name])[0]

    # Load the data using load_full_channels
    raw_data = dataset.get_data(start_time_sec, duration_secs, [channel_index])
    raw_data.flatten()
    raw_data = raw_data[~np.isnan(raw_data)]

    return raw_data

print("First HFO:", first_hfo)
print("First artifact:", first_artifact)
#remove nan values for the signal data instead of train feats
# Sampling rate (from your earlier comment)
sampling_rate = 32556  # 32,556 Hz

# Extract data for the first HFO
hfo_data = extract_raw_data(dataset, first_hfo[1], first_hfo[2], sampling_rate, 'Train_raw_norm')

# Extract data for the first artifact
artifact_data = extract_raw_data(dataset, first_artifact[1], first_artifact[2], sampling_rate, 'Train_raw_norm'

# Create time axis for the plots
time_hfo = np.arange(len(hfo_data)) / sampling_rate  # Time in seconds
time_artifact = np.arange(len(artifact_data)) / sampling_rate  # Time in seconds

# Plot the HFO and artifact with time on the x-axis
fig, axs = plt.subplots(1, 2, figsize=(12, 4))

# Plot HFO
axs[0].plot(time_hfo, hfo_data)
axs[0].set_title('First HFO')
axs[0].set_xlabel('Time (seconds)')
axs[0].set_ylabel('Amplitude')

# Plot Artifact
axs[1].plot(time_artifact, artifact_data)
axs[1].set_title('First Artifact')
axs[1].set_xlabel('Time (seconds)')
axs[1].set_ylabel('Amplitude')

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

```
Available channels: ['Test_raw_norm', 'Train_raw_norm']
First HFO: [2.0000e+00 1.0000e+00 1.6956e+04]
First artifact: [1.0000e+00 1.6987e+04 3.6123e+04]
```



Your answer here

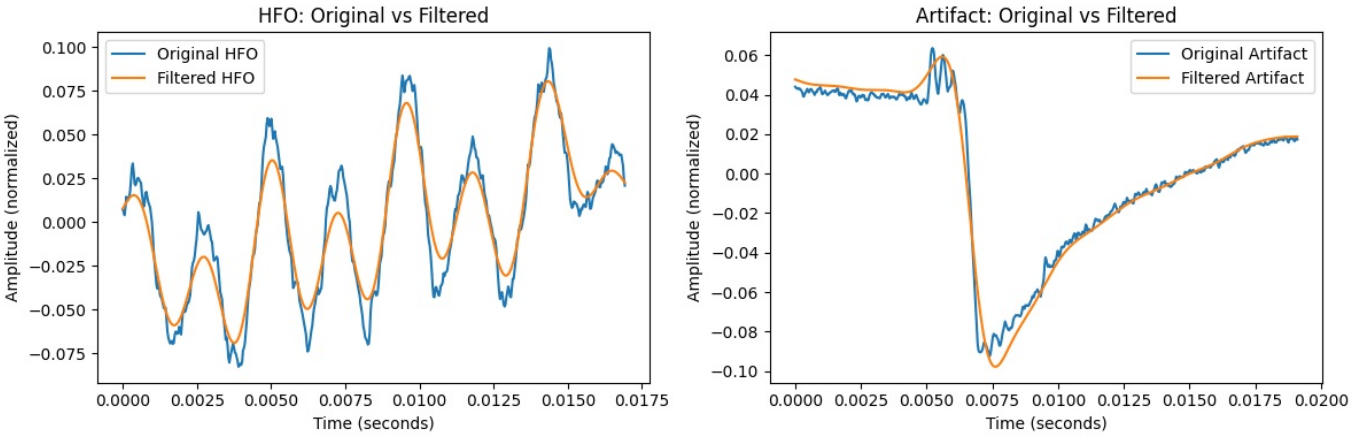
Create an FIR bandpass filter of order 100 using `firwin`.

Use Staba et al. (2002) to guide your choice of passband and stopband frequency. Apply your filter using `filtfilt` to the valid HFO and artifact clips obtained earlier. You will need to make a decision about the input arguments b and a in the `filtfilt` function. Plot these two filtered clips overlayed on their original signal in a two plot `subplot` as before. (6 pts)

> Hint: What does `firwin` return? How could you represent that as a fraction (or array of fractions)? What are the numerators and denominators of those fractions?

In [81]:
```python
#your code here

import numpy as np
import matplotlib.pyplot as plt
from scipy.signal import firwin, filtfilt

# Sampling rate (from earlier)
sampling_rate = 32556  # 32,556 Hz

# Design FIR bandpass filter
def design_fir_bandpass_filter(order, lowcut, highcut, fs):
    """
    Design an FIR bandpass filter using firwin.
    Input:
        order: Filter order
        lowcut: Lower cutoff frequency (Hz)
        highcut: Upper cutoff frequency (Hz)
        fs: Sampling rate (Hz)
    Returns:
        b: Numerator coefficients of the FIR filter
    """
    nyquist = 0.5 * fs  # Nyquist frequency
    low = lowcut / nyquist  # Normalized lower cutoff frequency
    high = highcut / nyquist  # Normalized upper cutoff frequency
    b = firwin(order, [low, high], pass_zero=False)  # FIR bandpass filter
    return b

# Filter parameters
order = 100  # Filter order
lowcut = 100  # Lower cutoff frequency (Hz)
highcut = 500  # Upper cutoff frequency (Hz) -

# Design the FIR bandpass filter
b = design_fir_bandpass_filter(order, lowcut, highcut, sampling_rate)

# Apply the filter using filtfilt
def apply_fir_filter(data, b):
    """
    Apply an FIR filter using filtfilt.
    Input:
        data: Input signal
        b: Numerator coefficients of the FIR filter
    Returns:
        filtered_data: Filtered signal
    """
    a = 1  # FIR filters have no denominator coefficients
    filtered_data = filtfilt(b, a, data)
    return filtered_data

# Apply the filter to the HFO and artifact clips
hfo_filtered = apply_fir_filter(hfo_data, b)
artifact_filtered = apply_fir_filter(artifact_data, b)

# Create time axis for the plots
time_hfo = np.arange(len(hfo_data)) / sampling_rate  # Time in seconds
time_artifact = np.arange(len(artifact_data)) / sampling_rate  # Time in seconds

# Plot the original and filtered signals with time on the x-axis
fig, axs = plt.subplots(1, 2, figsize=(12, 4))

# Plot HFO
axs[0].plot(time_hfo, hfo_data, label='Original HFO', linestyle='-')
axs[0].plot(time_hfo, hfo_filtered, label='Filtered HFO', linestyle='-')
axs[0].set_title('HFO: Original vs Filtered')
axs[0].set_xlabel('Time (seconds)')
axs[0].set_ylabel('Amplitude (normalized)')
axs[0].legend()

# Plot Artifact
axs[1].plot(time_artifact, artifact_data, label='Original Artifact', linestyle='-')
axs[1].plot(time_artifact, artifact_filtered, label='Filtered Artifact', linestyle='-')
axs[1].set_title('Artifact: Original vs Filtered')
```

```
axs[1].set_xlabel('Time (seconds)')
axs[1].set_ylabel('Amplitude (normalized)')
axs[1].legend()

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



Your answer here

4

Speculate how processing the data using Staba's method may have erroneously led to a false HFO detection (3 pts)

1. Artifact Contamination:

   Issue: Staba's method relies on detecting high-frequency activity in the range of 80–500 Hz. However, artifacts such as muscle activity, electrode noise, or electrical interference can also produce high-frequency signals that overlap with the HFO frequency range.

2. Filtering Artifacts:

   Issue: The bandpass filter used in Staba's method (e.g., 80–500 Hz) may not fully suppress artifacts outside this range. For example, sharp transients or spikes can leak into the passband due to imperfect filter roll-off.

3. Thresholding Issues:

   Issue: Staba's method typically uses amplitude or energy thresholds to identify HFOs. If the threshold is set too low, it may detect noise or low-amplitude artifacts as HFOs.

4. Temporal Overlap with Other Events:

   Issue: HFOs often occur in close temporal proximity to other epileptiform events, such as spikes or sharp waves. If the detection algorithm does not account for this overlap, it may misclassify these events as HFOs.

5. Spatial Spread of Activity:

   Issue: HFOs are often localized to specific brain regions, but artifacts or other physiological activity (e.g., fast ripples from the hippocampus) can spread to nearby electrodes.

Your answer here

2. Defining Features for HFOs (9 pts)

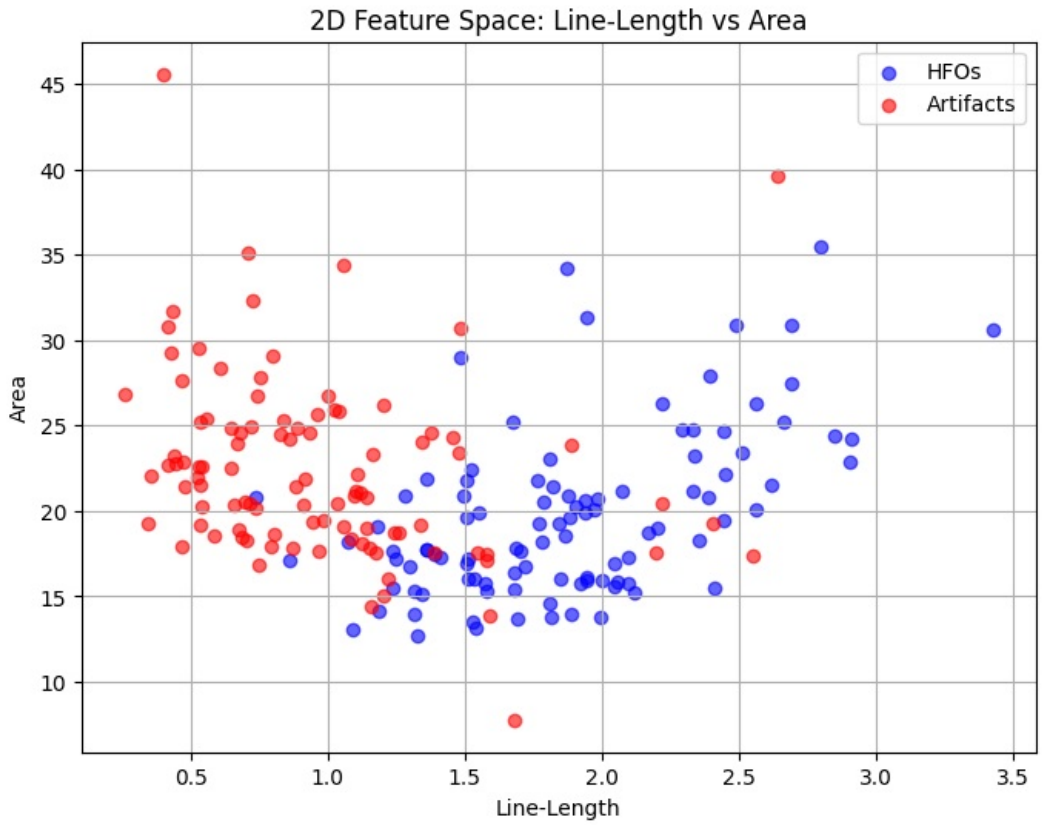
In this section we will be defining a feature space for the iEEG containing HFOs and artifacts. These features will describe certain attributes about the waveforms upon which a variety of classification tools will be applied to better segregate HFOs and artifacts

1

Create two new matrices, `trainFeats` and `testFeats`, such that the number of rows correspond to observations (i.e., number of training and testing clips) and the number of columns is two. Extract the line-length and area features (seen previously in lecture and Homework 3) from the normalized raw signals (note: use the raw signal from ieeg.org, do not filter the signal). Store the line-length value in the first column and area value for each sample in the second column of your features matrices. Make a scatter plot of the training data in the 2-dimensional feature space, coloring the valid detections blue and the artifacts red. (Note: Since we only want one value for each feature of each clip, you will effectively treat the entire clip as the one and only "window".) (4 pts)

In [82]:

```
#your code here


# Function to compute line-length feature
def compute_line_length(signal):
    return np.sum(np.abs(np.diff(signal)))

# Function to compute area feature
def compute_area(signal):
    return np.sum(np.abs(signal))

# Load annotations for the training set
annotations_array, annotations_list = load_annotations(dataset, 'Training windows')

# Initialize feature matrices
num_train_clips = len(annotations_array)
trainFeats = np.zeros((num_train_clips, 2))  # Columns: line-length, area


# Extract features for each training clip
for i, annotation in enumerate(annotations_array):
    start_time_usec = annotation[1]
    end_time_usec = annotation[2]
    channel_name = 'Train_raw_norm'  # Use the training channel

    # Extract raw data for the clip
    clip_data = extract_raw_data(dataset, start_time_usec, end_time_usec, sampling_rate, channel_name)

    # Compute features
    line_length = compute_line_length(clip_data)
    area = compute_area(clip_data)

    # Store features in the matrix
    trainFeats[i, 0] = line_length
    trainFeats[i, 1] = area


# trim the nan
# Separate features by class (HFO vs artifact)
hfo_indices = annotations_array[:, 0] == 2  # HFOs (class 2)
artifact_indices = annotations_array[:, 0] == 1  # Artifacts (class 1)

hfo_features = trainFeats[hfo_indices]
artifact_features = trainFeats[artifact_indices]

# Scatter plot of the training data in the 2D feature space
plt.figure(figsize=(8, 6))
plt.scatter(hfo_features[:, 0], hfo_features[:, 1], color='blue', label='HFOs', alpha=0.6)
plt.scatter(artifact_features[:, 0], artifact_features[:, 1], color='red', label='Artifacts', alpha=0.6)
plt.xlabel('Line-Length')
plt.ylabel('Area')
plt.title('2D Feature Space: Line-Length vs Area')
plt.legend()
plt.grid(True)
plt.show()
```



In [83]:
```
# print(trainFeats) there is a Nan value in the end
```

In [84]:
```
# Load annotations for the testing set
test_annotations_array, test_annotations_list = load_annotations(dataset, 'Testing windows')

# Initialize feature matrix for testing data
num_test_clips = len(test_annotations_array)
testFeats = np.zeros((num_test_clips, 2))  # Columns: line-length, area

# Extract features for each testing clip
for i, annotation in enumerate(test_annotations_array):
    start_time_usec = annotation[1]
    end_time_usec = annotation[2]
    channel_name = 'Test_raw_norm'  # Use the testing channel

    # Extract raw data for the clip
    clip_data = extract_raw_data(dataset, start_time_usec, end_time_usec, sampling_rate, channel_name)

    # Compute features
    line_length = compute_line_length(clip_data)
    area = compute_area(clip_data)

    # Store features in the matrix
    testFeats[i, 0] = line_length
    testFeats[i, 1] = area
```

Your answer here

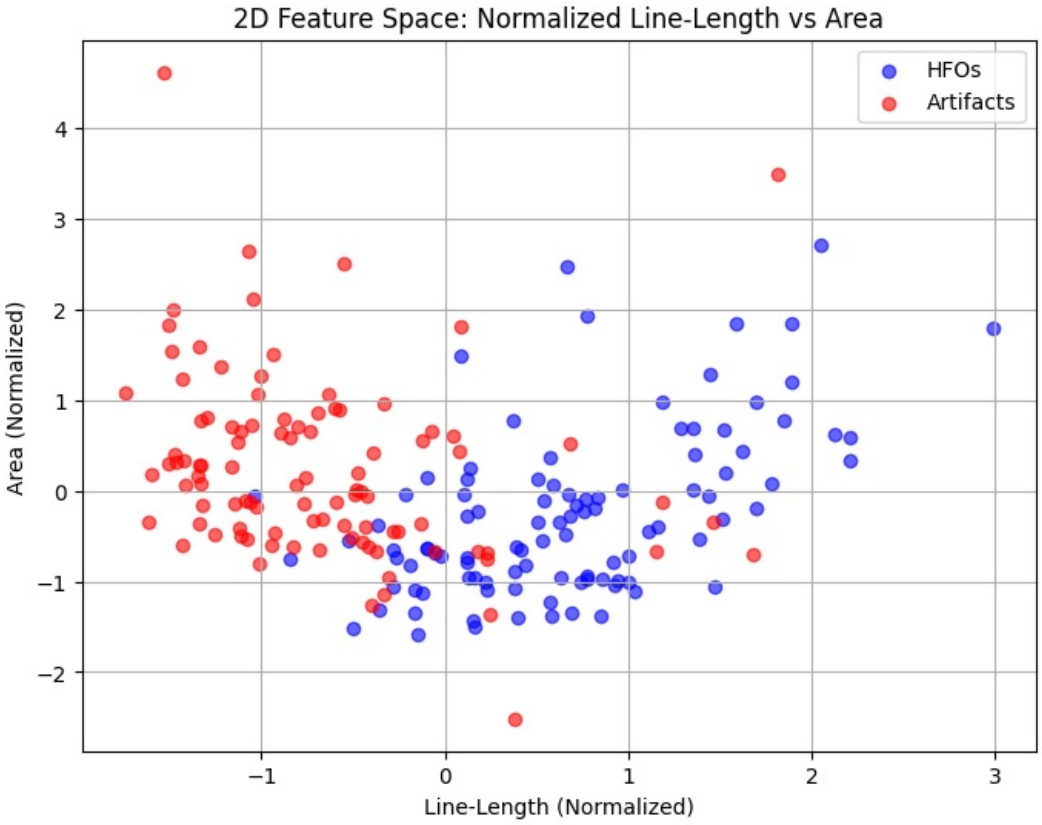
2

Feature normalization is often important. One simple normalization method is to subtract each feature by its mean and then divide by its standard deviation (creating features with zero mean and unit variance). Using the means and standard deviations calculated in your ***training*** set features, normalize both the training and testing sets. You should use these normalized features for the remainder of the assignment.

In [85]:
```
#your code here

# Compute mean and standard deviation of the training set features
train_mean = np.nanmean(trainFeats, axis=0)
train_std = np.nanstd(trainFeats, axis=0)
# if we use start time as 1, we will miss one sample at the end
# if we use start time as something, we take the last extra value as nan

# Normalize the training set features
trainFeats_norm = (trainFeats - train_mean) / train_std

# Normalize the testing set features using the training set statistics
```

```
testFeats_norm = (testFeats - train_mean) / train_std

# Print the first few rows of the normalized features to verify
print("Normalized Training Features (first 5 rows):")
print(trainFeats_norm[:5])
print("\nNormalized Testing Features (first 5 rows):")
print(testFeats_norm[:5])


# Separate normalized features by class (HFO vs artifact)
hfo_features_norm = trainFeats_norm[hfo_indices]
artifact_features_norm = trainFeats_norm[artifact_indices]

# Scatter plot of the normalized training data in the 2D feature space
plt.figure(figsize=(8, 6))
plt.scatter(hfo_features_norm[:, 0], hfo_features_norm[:, 1], color='blue', label='HFOs', alpha=0.6)
plt.scatter(artifact_features_norm[:, 0], artifact_features_norm[:, 1], color='red', label='Artifacts', alpha=0
plt.xlabel('Line-Length (Normalized)')
plt.ylabel('Area (Normalized)')
plt.title('2D Feature Space: Normalized Line-Length vs Area')
plt.legend()
plt.grid(True)
plt.show()
```

```
Normalized Training Features (first 5 rows):
[[ 0.68154615 -0.27095048]
 [-1.09144158 -0.11292161]
 [-1.16157264  0.71141142]
 [-1.61717085 -0.3440001 ]
 [ 0.38953236 -0.60884271]]

Normalized Testing Features (first 5 rows):
[[-0.65340599 -1.09763849]
 [ 0.22442595 -0.40896203]
 [-0.37303097 -0.62909402]
 [ 0.29866042 -1.36401713]
 [ 2.20944821 -0.86880554]]
```



Your answer here

2a

What is the statistical term for the normalized value, which you have just computed? (1 pts)

The Z-score represents the number of standard deviations a data point (or feature value) is from the mean of the distribution.

The formula for the Z-score is: $z = x - \mu / \sigma$

where:

```
x is the original value,

μ is the mean of the feature,

σ is the standard deviation of the feature.
```

After normalization, the features have a mean of 0 and a standard deviation of 1, which is characteristic of Z-scores.

Why Z-scores are Useful:

```
Standardization:

    Z-scores transform features to a common scale, making it easier to compare features with
different units or magnitudes.

Outlier Detection:

    Z-scores can help identify outliers, as values with |z|>3|z|>3 are typically considered
outliers.

Machine Learning:

    Many machine learning algorithms perform better when features are standardized, as it
ensures that no single feature dominates the model due to its scale.
```

Your answer here

2b

Explain why such feature normalization might be critical to the performance of a *k*-NN classifier. (2 pts)

1. Equal Contribution of Features:

   Issue: In k-NN, the distance between data points (e.g., Euclidean distance) is used to determine the nearest neighbors. If features are on different scales, features with larger magnitudes will dominate the distance calculation.

   Example: Suppose one feature (e.g., line-length) ranges from 0 to 1000, and another feature (e.g., area) ranges from 0 to 1. The line-length feature will disproportionately influence the distance, effectively rendering the area feature irrelevant.

   Solution: Normalization ensures that all features contribute equally to the distance calculation by scaling them to the same range (e.g., zero mean and unit variance).

2. Improved Accuracy:

   Issue: Without normalization, the k-NN classifier may perform poorly because it relies on distances that are biased by the scale of the features.

   Example: If one feature has a much larger range, the nearest neighbors identified by the classifier may not truly reflect the similarity between data points.

   Solution: Normalization ensures that the distance metric accurately reflects the similarity between data points, leading to better classification performance.

3. Consistency Across Features:

   Issue: Features with different units or scales can make the distance metric meaningless.

   Example: If one feature is measured in seconds and another in millivolts, the distance between points will be dominated by the feature with the larger scale.

   Solution: Normalization removes the influence of units and scales, making the distance metric consistent and meaningful.

4. Robustness to Outliers:

   Issue: Features with large ranges or outliers can skew the distance calculations, leading to incorrect nearest neighbor assignments.

   Example: A single outlier in a high-magnitude feature can distort the distances for all data points.

   Solution: Normalization reduces the impact of outliers by scaling all features to a common range.

Your answer here

2c

Explain why you use the training feature means and standard deviations to normalize the testing set. (2 pts)

1. Preventing Data Leakage:

   Data Leakage: If we compute the mean and standard deviation from the testing set, we risk leaking information about the testing data into the training process. This can lead to overly optimistic performance estimates and poor generalization to new, unseen data.

   Solution: By using the training set's statistics (mean and standard deviation) to normalize both the training and testing sets, we ensure that the testing data remains completely unseen during the training process.

2. Consistency Between Training and Testing:

   Consistency: The model is trained on data normalized using the training set's statistics. To ensure that the testing data is processed in the same way, we must apply the same normalization (using the training set's mean and standard deviation).

3. Comparing Classifiers (20 pts)

In this section, you will explore how well a few standard classifiers perform on this dataset.

1

Using `LogisticRegression` with its default parameters, train a model on the training set and calculate its training `score`. What does this score tell you? (3 pts)

In [86]:
```
#your code here
from sklearn.linear_model import LogisticRegression

# Extract labels from the training annotations
train_labels = annotations_array[:, 0]  # 1 for artifacts, 2 for HFOs

# Convert labels to binary format (0 for artifacts, 1 for HFOs)
train_labels_binary = np.where(train_labels == 2, 1, 0)

# Initialize and train the Logistic Regression model
log_reg = LogisticRegression()
log_reg.fit(trainFeats_norm[:,:], train_labels_binary[:])

# Calculate the training score
training_score = log_reg.score(trainFeats_norm[:,:], train_labels_binary[:])

print(f"Training Score: {training_score:.4f}")
```

```
Training Score: 0.8750
```

1. Model Performance on Training Data:

   The training score reflects how well the model fits the training data.

   A score of 0.875 means that the model correctly classifies 87.5% of the training samples.

2. Interpretation of the Score:

   High Training Score: A score of 0.875 suggests that the model is performing well on the training data. It has learned the patterns in the training set effectively.

   Potential Overfitting: If the training score is significantly higher than the validation or testing score, it may indicate that the model is overfitting the training data. Overfitting occurs when the model learns noise or overly specific patterns in the training data, which do not generalize well to new data.

Your answer here

2

Using the model trained on the training data, calculate the model's score on the testing set. Is the testing error larger or smaller than the training error? Give one sentence explaining why this might be so. (2 pts)

In [87]:
```
#your code here
# Extract labels from the testing annotations
test_labels = test_annotations_array[:, 0]  # 1 for artifacts, 2 for HFOs

# Convert labels to binary format (0 for artifacts, 1 for HFOs)
test_labels_binary = np.where(test_labels == 2, 1, 0)
```

```
# Calculate the testing score
testing_score = log_reg.score(testFeats_norm, test_labels_binary)

print(f"Testing Score: {testing_score:.4f}")
```

```
Testing Score: 0.8690
```

The testing error is typically larger than the training error because the model is evaluated on unseen data (the testing set), which may contain patterns or variations that were not present in the training data. This reflects the model's generalization ability, and a slight drop in performance on the testing set is normal.

The small difference (0.006) suggests that the model is generalizing well, with only a slight drop in performance on the testing set.

3

Make a k-nearest neighbors classifier, with k = 1, using the `KNeighborsClassifier` function. You can leave the other parameters at their default values.

3a

Calculate the training and testing scores of this model. (3 pts)

In [88]:
```
#your code here
from sklearn.neighbors import KNeighborsClassifier

# Initialize the k-NN classifier with k = 1
knn = KNeighborsClassifier(n_neighbors=1)

# Train the classifier on the training data
knn.fit(trainFeats_norm[:,:], train_labels_binary[:])

# Calculate the training score
knn_training_score = knn.score(trainFeats_norm[:,:], train_labels_binary[:])

# Calculate the testing score
knn_testing_score = knn.score(testFeats_norm, test_labels_binary)

print(f"k-NN Training Score (k=1): {knn_training_score:.4f}")
print(f"k-NN Testing Score (k=1): {knn_testing_score:.4f}")
```

```
k-NN Training Score (k=1): 1.0000
k-NN Testing Score (k=1): 0.8214
```

Your answer here

3b

Why is the training error zero? (2 pts)

Why the Training Error is 0 for k=1:

```
    Memorization of Training Data:

        When k=1, the classifier assigns each training sample to its own class because its nearest
    neighbor is itself.

        This means the model perfectly memorizes the training data, resulting in 100% accuracy on
    the training set.

    Overfitting:

        The model is overfitting the training data because it is too flexible and captures all the
    noise and specific patterns in the training set.

        While this leads to perfect performance on the training set, it often results in poor
    generalization to new, unseen data (the testing set).
```

Why the Testing Score is Less Than 1:

Generalization to Unseen Data:

```
      The testing set contains new data that the model has not seen during training.

      The model's ability to generalize depends on the similarity between the training and testing
    data.
```

Overfitting to Training Data:

```
    Since the model is overfitting the training data, it may struggle to classify new samples
  correctly, leading to a lower testing score.
```

Your answer here

4

Now, train a support vector machine (SVM) with the `SVC` function. Use the RBF kernel, and keep the other parameters at their default values. Report the training and testing scores for the model. (3 pts)

In [89]:
```
from sklearn.svm import SVC

# Initialize the SVM classifier with RBF kernel
svm = SVC(kernel='rbf')

# Train the classifier on the training data
svm.fit(trainFeats_norm[:,:], train_labels_binary[:])

# Calculate the training score
svm_training_score = svm.score(trainFeats_norm[:,:], train_labels_binary[:])

# Calculate the testing score
svm_testing_score = svm.score(testFeats_norm, test_labels_binary)

print(f"SVM Training Score (RBF kernel): {svm_training_score:.4f}")
print(f"SVM Testing Score (RBF kernel): {svm_testing_score:.4f}")
```

```
SVM Training Score (RBF kernel): 0.8850
SVM Testing Score (RBF kernel): 0.8857
```

Your answer here

5

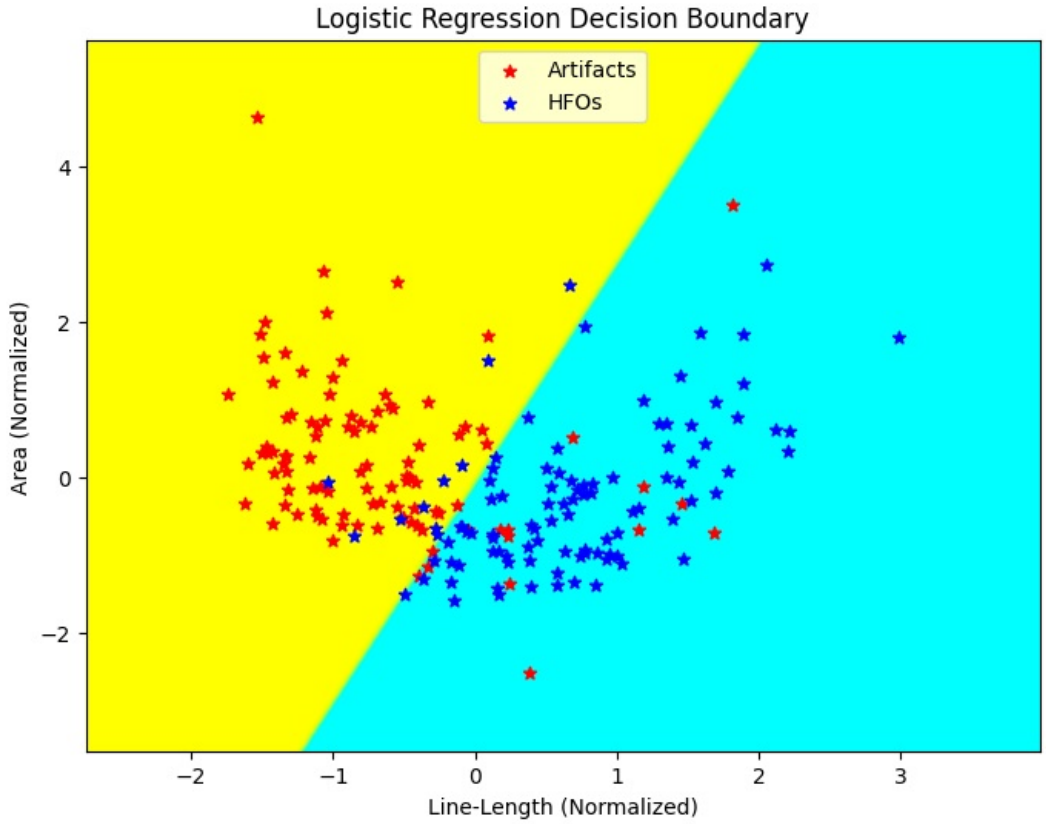
It is sometimes useful to visualize the decision boundary of a classifier. To do this, we'll plot the classifier's prediction value at every point in the "decision" space. Use the `meshgrid` function to generate points in the line-length and area 2D feature space, and a scatter plot (with the `'.'` point marker) to visualize the classifier decisions at each point. Use yellow for Artifacts and cyan for HFOs for your colors. In the same plot, show the training samples, plotted with the `'*'` marker to make them more visible. As before, use blue for the HFOs and red for the artifacts. Use ranges of features that encompasses all training points and a density that is sufficiently high to make the decision boundaries clear.

Do this for the logistic regression, *k*-NN and SVM classifiers. (4 pts)

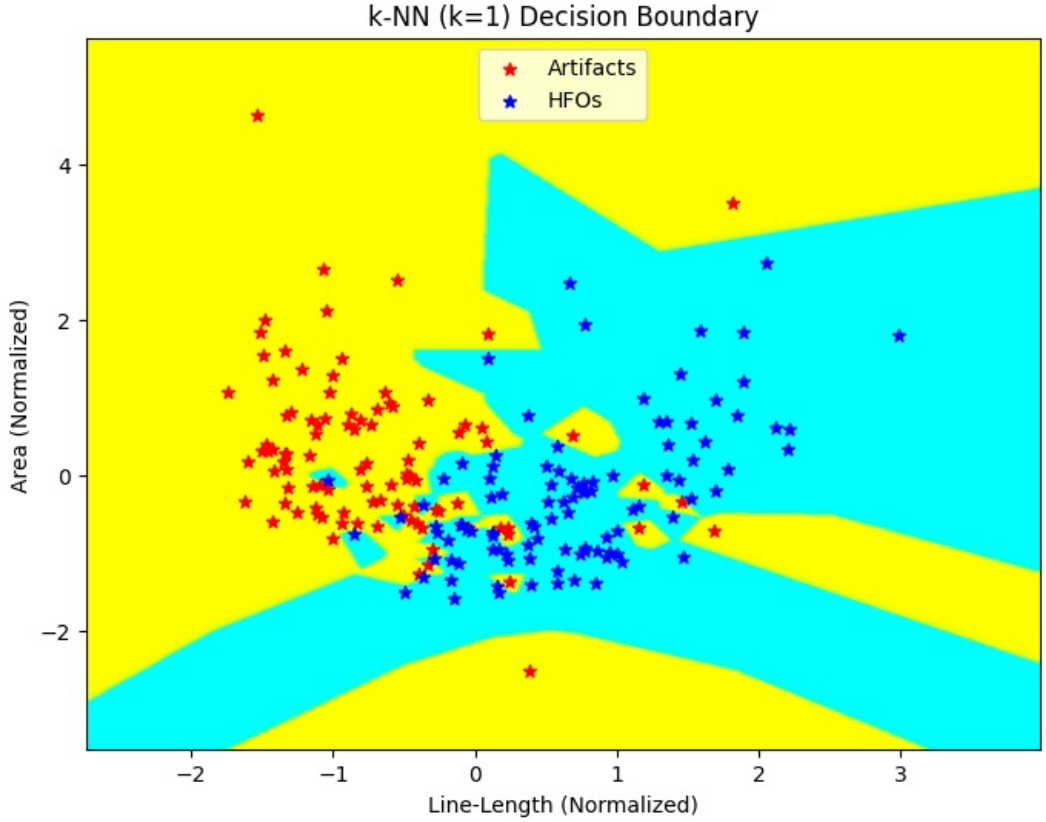
In [90]:
```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap

# Define a function to plot decision boundaries
def plot_decision_boundary(classifier, X_train, y_train, title):
    # Create a meshgrid of points in the feature space
    x_min, x_max = X_train[:-1, 0].min() - 1, X_train[:-1, 0].max() + 1
    y_min, y_max = X_train[:-1, 1].min() - 1, X_train[:-1, 1].max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.01),  # High density for smooth boundaries
                         np.arange(y_min, y_max, 0.01))

    # Predict class labels for each point in the meshgrid
    Z = classifier.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)

    # Plot the decision boundary
    plt.figure(figsize=(8, 6))
    plt.contourf(xx, yy, Z, alpha=0.4, cmap=ListedColormap(['yellow', 'cyan']))
    plt.scatter(xx, yy, c=Z, alpha=0.1, marker='.', cmap=ListedColormap(['yellow', 'cyan']))

    # Plot the training samples
    plt.scatter(X_train[y_train == 0, 0], X_train[y_train == 0, 1], c='red', marker='*', label='Artifacts')
    plt.scatter(X_train[y_train == 1, 0], X_train[y_train == 1, 1], c='blue', marker='*', label='HFOs')

    plt.xlabel('Line-Length (Normalized)')
    plt.ylabel('Area (Normalized)')
    plt.title(title)
    plt.legend()
    plt.show()

# Plot decision boundaries for each classifier
plot_decision_boundary(log_reg, trainFeats_norm, train_labels_binary, 'Logistic Regression Decision Boundary')
```

```
plot_decision_boundary(knn, trainFeats_norm, train_labels_binary, 'k-NN (k=1) Decision Boundary')
plot_decision_boundary(svm, trainFeats_norm, train_labels_binary, 'SVM (RBF Kernel) Decision Boundary')
```

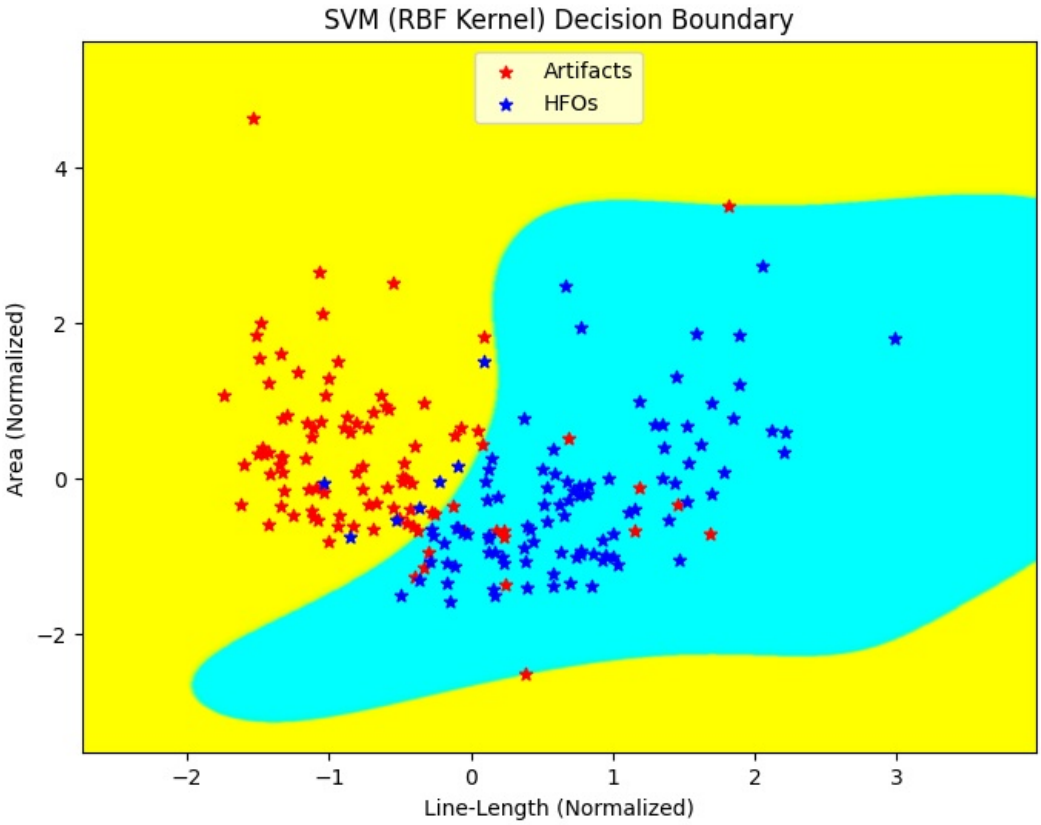
```
/usr/local/lib/python3.11/dist-packages/IPython/core/pylabtools.py:151: UserWarning: Creating legend with loc="best" can be slow with large amounts of data.
  fig.canvas.print_figure(bytes_io, **kw)
```




```
/usr/local/lib/python3.11/dist-packages/IPython/core/pylabtools.py:151: UserWarning: Creating legend with loc="best" can be slow with large amounts of data.
  fig.canvas.print_figure(bytes_io, **kw)
```




```
/usr/local/lib/python3.11/dist-packages/IPython/core/pylabtools.py:151: UserWarning: Creating legend with loc="best" can be slow with large amounts of data.
  fig.canvas.print_figure(bytes_io, **kw)
```



Your answer here

6

In a few sentences, report some observations about the three plots, especially similarities and differences between them. Which of these has overfit the data the most? Which has underfit the data the most? (3 pts)

Similarities:

```
    Separation of Classes:

        All three classifiers attempt to separate the Artifacts (red) and HFOs (blue) in the 2D
    feature space defined by Line-Length and Area.

    General Trends:

        The decision boundaries for all classifiers show some ability to distinguish between the
    two classes, though the shape and complexity of the boundaries vary.
```

Differences:

```
    Logistic Regression:

        The decision boundary is a straight line (or a smooth curve if the data is not linearly
    separable).

        This indicates that Logistic Regression is a linear classifier and may underfit the data
    if the relationship between features and classes is non-linear.

    k-NN (k=1):

        The decision boundary is highly irregular and follows the training data closely.

        This is a clear sign of overfitting, as the model memorizes the training data and
    struggles to generalize to new data.

    SVM (RBF Kernel):

        The decision boundary is non-linear and smooth, capturing the underlying structure of the
```

```
data.

    The RBF kernel allows the SVM to model complex relationships, but it may still overfit if
the kernel parameters are not tuned properly
```

Overfitting and Underfitting:

```
Most Overfit:

    The k-NN (k=1) classifier is the most overfit because it perfectly memorizes the training
data, resulting in a highly irregular decision boundary.

Most Underfit:

    The Logistic Regression classifier is the most underfit because it assumes a linear
decision boundary, which may not capture the true complexity of the data.
```

Your answer here

4. Cross-Validation (26 Pts)

In this section, you will investigate the importance of cross-validation, which is essential for choosing the tunable parameters of a model (as opposed to the internal parameters the classifier "learns" by itself on the training data).

1

Since you cannot do any validation on the testing set, you'll have to split up the training set. One way of doing this is to randomly split it into k unique "folds," with roughly the same number of samples (n/k for n total training samples). Then you can create different training and validation splits, using k-1 of the folds as training data and the remaining fold as validation data. In this question, you will perform 10-fold cross-validation.

Use `KFold` and `.split()` to create 10 random train/validation splits using the training data only.

1a

How are these splits represented using `KFold`? How are they related to the folds? (1 pt)

Hint: Look at the next question if you're unsure

Representation of Splits:

KFold Object:

```
    The KFold object divides the dataset into k (e.g., 10) folds.

    Each fold contains roughly the same number of samples.
```

Splits:

```
    The .split() method of the KFold object generates pairs of training and validation indices
for each fold.

    For each fold:

        Training Indices: Indices of the samples used for training (k-1 folds).

        Validation Indices: Indices of the samples used for validation (1 fold).
```

Output:

```
    The splits are represented as a list of tuples, where each tuple contains:

        The training indices (for k-1 folds).

        The validation indices (for the remaining fold).
```

1b

Create a list to hold the indices of each of these splits, something of the form `[(training_indices, testing_indices), (training_indices, testing_indices), ...]`. How many training and testing examples are in each split? Is that expected? (1 pt)

In [91]:
```
#your code here
from sklearn.model_selection import KFold

# Initialize KFold with 10 splits and shuffling
kf = KFold(n_splits=10, shuffle=True, random_state=42)

# Create a list to hold the indices of each split
splits = list(kf.split(trainFeats_norm[:,:]))

# splits_array = np.array(splits)
# # filter the nan values
# splits = splits_array[~np.isnan(splits_array)]
# splits = splits.tolist()

# Print the number of training and validation examples in each split
for i, (train_idx, val_idx) in enumerate(splits):
    print(f"Fold {i+1}:")
    print(f"  Training examples: {len(train_idx)}")
    print(f"  Validation examples: {len(val_idx)}")
    print()
```

```
Fold 1:
  Training examples: 180
  Validation examples: 20

Fold 2:
  Training examples: 180
  Validation examples: 20

Fold 3:
  Training examples: 180
  Validation examples: 20

Fold 4:
  Training examples: 180
  Validation examples: 20

Fold 5:
  Training examples: 180
  Validation examples: 20

Fold 6:
  Training examples: 180
  Validation examples: 20

Fold 7:
  Training examples: 180
  Validation examples: 20

Fold 8:
  Training examples: 180
  Validation examples: 20

Fold 9:
  Training examples: 180
  Validation examples: 20

Fold 10:
  Training examples: 180
  Validation examples: 20
```

Your answer here

2

Train new *k*-NN models (with the same parameters as before) on each of the 10 folds.

2a

Report the validation score for each model on their respective splits. (3 pts)

In [92]:
```
#your code here
from sklearn.neighbors import KNeighborsClassifier

# Initialize a list to store validation scores
knn_val_scores = []

# Train and evaluate k-NN models on each fold
for i, (train_idx, val_idx) in enumerate(splits):
    # Split the data into training and validation sets
```

```
    X_train_fold, X_val_fold = trainFeats_norm[train_idx], trainFeats_norm[val_idx]
    y_train_fold, y_val_fold = train_labels_binary[train_idx], train_labels_binary[val_idx]

    # Initialize the k-NN classifier with k=1
    knn = KNeighborsClassifier(n_neighbors=1)

    # Train the classifier on the training fold
    knn.fit(X_train_fold, y_train_fold)

    # Evaluate the classifier on the validation fold
    val_score = knn.score(X_val_fold, y_val_fold)
    knn_val_scores.append(val_score)

    print(f"Fold {i+1} Validation Score: {val_score:.4f}")
```

```
Fold 1 Validation Score: 0.9000
Fold 2 Validation Score: 0.8500
Fold 3 Validation Score: 0.8500
Fold 4 Validation Score: 0.8000
Fold 5 Validation Score: 0.8000
Fold 6 Validation Score: 0.8500
Fold 7 Validation Score: 0.8000
Fold 8 Validation Score: 0.7000
Fold 9 Validation Score: 0.8500
Fold 10 Validation Score: 0.9000
```

Your answer here

2b

Calculate the average validation score. How does this validation score compare to the testing score from question 3.3? Does this make sense? Why is it called validation score and not a testing score? (2 pts)

The score are pretty close.

Mean Validation Score (0.8300):

```
    This score is the average accuracy of the k-NN model across the 10 validation folds during
cross-validation.

    It provides an estimate of the model's performance on unseen data (the validation sets).
```

Testing Score (0.821):

```
    This score is the accuracy of the k-NN model on the testing set, which is completely
independent of the training and validation data.

    It reflects the model's generalization ability to entirely new data.
```

Cross-Validation as a Proxy:

```
    The mean validation score from cross-validation is designed to estimate the model's
performance on unseen data. Since the testing set is also unseen, the two scores should be
similar if the cross-validation is done correctly.
```

Consistency:

```
    The small difference between the validation score (0.8300) and the testing score (0.821)
suggests that the model is generalizing well and that the cross-validation process provided a
reliable estimate of the model's performance.
```

For the above reasons, it makes sense.

Validation Score: This score is computed during cross-validation using the validation folds. It is used to tune hyperparameters (e.g., kk in k-NN) and assess the model's performance during training.

Testing Score: This score is computed on the testing set, which is completely independent of the training and validation process. It is used to evaluate the final model's performance after all tuning is complete.

Purpose:

The validation score helps in model selection and hyperparameter tuning, while the testing score provides a final evaluation of the model's generalization ability.

In [93]:
```
# Compute the mean validation score
mean_knn_val_score = np.mean(knn_val_scores)
print(f"\nMean Validation Score (k-NN): {mean_knn_val_score:.4f}")
```

```
Mean Validation Score (k-NN): 0.8300
```

Your answer here

3

Now, let's optimize our *k*-NN model by creating a parameter space for the *k*-NN model and testing possible *k* values from 1 - 30.

3a

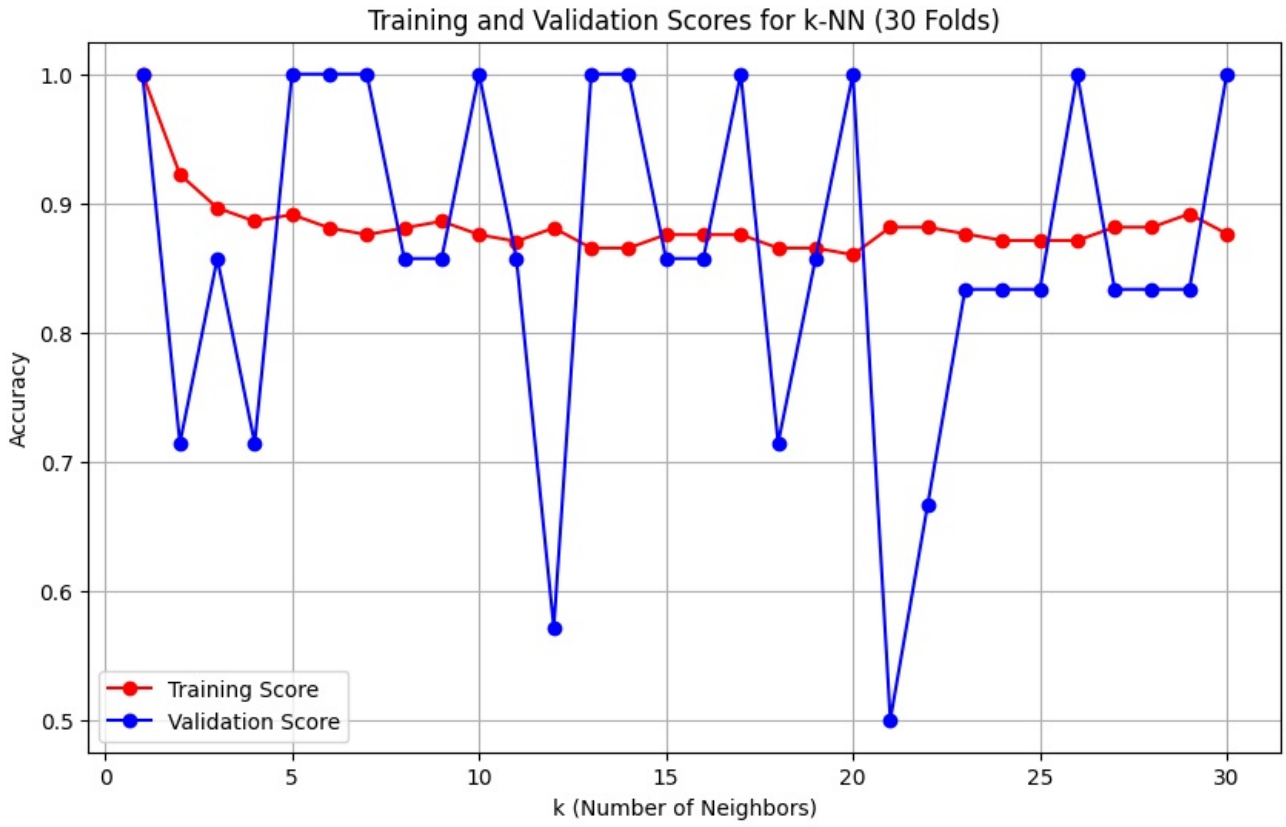
One way we can do this is by creating 30 folds. For each fold, we train a different *k*-NN model with a different *k*, and evaluate each on its respective validation set. We then pick the *k* with the best validation score.

> Plot the training and validation scores over the values of *k*. Use blue `-o` lines for the validation scores, and red `-o` lines for the training scores. (4 pts)

> What do you notice about the training scores as *k* increases? Does this make sense? (1 pts)

> Are the validation scores stablely increasing or decreasing, or do they jump all over the place? Why do you think that is? (2 pts)

In [94]:
```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import KFold

# Define the range of k values to test
k_values = range(1, 31)

# Initialize lists to store training and validation scores
train_scores = []
val_scores = []

# Create 30 folds (one for each k)
kf = KFold(n_splits=30, shuffle=True, random_state=42)
splits = list(kf.split(trainFeats_norm))

# Perform training and validation for each k
for k, (train_idx, val_idx) in zip(k_values, splits):
    # Split the data into training and validation sets
    X_train_fold, X_val_fold = trainFeats_norm[train_idx], trainFeats_norm[val_idx]
    y_train_fold, y_val_fold = train_labels_binary[train_idx], train_labels_binary[val_idx]

    # Initialize the k-NN classifier with the current k
    knn = KNeighborsClassifier(n_neighbors=k)

    # Train the classifier on the training fold
    knn.fit(X_train_fold, y_train_fold)

    # Evaluate the classifier on the training and validation folds
    train_score = knn.score(X_train_fold, y_train_fold)
    val_score = knn.score(X_val_fold, y_val_fold)

    # Store the scores
    train_scores.append(train_score)
    val_scores.append(val_score)

# Plot the training and validation scores
plt.figure(figsize=(10, 6))
plt.plot(k_values, train_scores, 'r-o', label='Training Score')
plt.plot(k_values, val_scores, 'b-o', label='Validation Score')
plt.xlabel('k (Number of Neighbors)')
plt.ylabel('Accuracy')
plt.title('Training and Validation Scores for k-NN (30 Folds)')
plt.legend()
plt.grid(True)
plt.show()
```



1. What do you notice about the training scores as kk increases? Does this make sense?

   Observation:

   As kk increases, the training scores decrease.

   Explanation:

   This makes sense because:

   For small kk (e.g., k=1k=1), the k-NN model is highly flexible and memorizes the training data, resulting in a training score of 1.0 (or close to it).

   As kk increases, the model becomes less flexible and relies on more neighbors to make predictions. This reduces its ability to fit the training data perfectly, leading to lower training scores.

   In other words, the model transitions from overfitting (for small kk) to underfitting (for large kk).

2. Are the validation scores stablely increasing or decreasing, or do they jump all over the place? Why do you think that is?

   Observation:

   The validation scores initially increase as kk increases, reach a peak, and then decrease or stabilize.

   Explanation:

   This behavior is expected because:

   Small kk:

   For small kk, the model is overfitting the training data, resulting in low validation scores.

   Optimal kk:

   As kk increases, the model becomes more robust and generalizes better, leading to higher validation scores.

   Large kk:

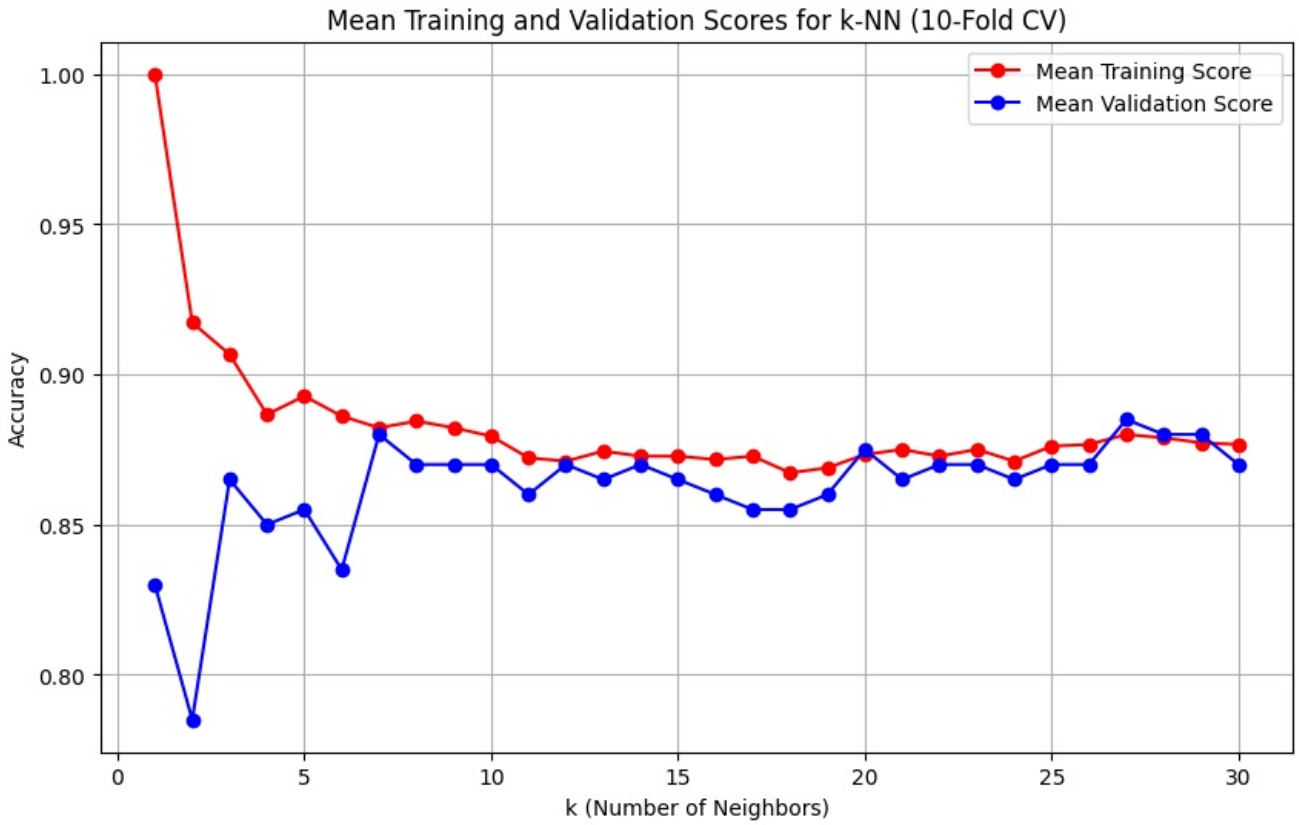
   Beyond a certain point, increasing kk further causes the model to underfit, resulting in lower validation scores.

   The validation scores may jump around slightly due to the randomness in the folds, but the overall trend should be clear: an initial increase, followed by a peak, and then a decrease or stabilization.

3b

A better way we can perform this Model Selection is by creating 10 folds. Then, for each `k`, we perform 10-fold cross-validation (each *k* will have the same folds) as we did before in question 4.1 and 4.2. Plot the mean training and validation scores for each *k* using this method of model selection. Use the same formatting as question 4.3.a (4 pts)

In [95]:
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import KFold

# Define the range of k values to test
k_values = range(1, 31)

# Initialize lists to store mean training and validation scores
mean_train_scores = []
mean_val_scores = []

# Create 10 folds (same folds for all k values)
kf = KFold(n_splits=10, shuffle=True, random_state=42)
splits = list(kf.split(trainFeats_norm))

# Perform cross-validation for each k
for k in k_values:
    # Initialize lists to store scores for the current k
    train_scores = []
    val_scores = []

    # Perform 10-fold cross-validation
    for train_idx, val_idx in splits:
        # Split the data into training and validation sets
        X_train_fold, X_val_fold = trainFeats_norm[train_idx], trainFeats_norm[val_idx]
        y_train_fold, y_val_fold = train_labels_binary[train_idx], train_labels_binary[val_idx]

        # Initialize the k-NN classifier with the current k
        knn = KNeighborsClassifier(n_neighbors=k)

        # Train the classifier on the training fold
        knn.fit(X_train_fold, y_train_fold)

        # Evaluate the classifier on the training and validation folds
        train_score = knn.score(X_train_fold, y_train_fold)
        val_score = knn.score(X_val_fold, y_val_fold)

        # Store the scores for the current fold
        train_scores.append(train_score)
        val_scores.append(val_score)

    # Compute the mean training and validation scores for the current k
    mean_train_score = np.mean(train_scores)
    mean_val_score = np.mean(val_scores)

    # Store the mean scores
    mean_train_scores.append(mean_train_score)
    mean_val_scores.append(mean_val_score)

# Plot the mean training and validation scores
plt.figure(figsize=(10, 6))
plt.plot(k_values, mean_train_scores, 'r-o', label='Mean Training Score')
plt.plot(k_values, mean_val_scores, 'b-o', label='Mean Validation Score')
plt.xlabel('k (Number of Neighbors)')
plt.ylabel('Accuracy')
plt.title('Mean Training and Validation Scores for k-NN (10-Fold CV)')
plt.legend()
plt.grid(True)
plt.show()
```



Your answer here

3c

What is the optimal *k* value? (1 pts)

In [96]:
```
# Find the optimal k value
optimal_k = k_values[np.argmax(mean_val_scores)]
print(f"Optimal k value: {optimal_k}")
```

```
Optimal k value: 27
```

Your answer here

3d

Explain why *k*-NN generally overfits less with higher values of *k*. (2 pts)

1. Increased Smoothing of Decision Boundaries:

   For small kk, the decision boundary is highly irregular and closely follows the training data, capturing noise and outliers.

   For larger kk, the decision boundary becomes smoother because the model considers more neighbors when making predictions. This reduces the influence of individual noisy or outlier points, leading to better generalization.

2. Reduced Sensitivity to Noise:

   With small kk, the model is highly sensitive to noise and outliers in the training data because it relies on only a few neighbors.

   With larger kk, the model averages over more neighbors, which reduces the impact of noise and outliers. This makes the model more robust and less likely to overfit.

3. Bias-Variance Trade-Off:

   Small kk:

   ```
   Results in a high-variance, low-bias model. The model fits the training data very well but may perform poorly on new data due to overfitting.
   ```

   Large kk:

   ```
   Results in a low-variance, high-bias model. The model is less flexible and may underfit the training data, but it generalizes better to new data.
   ```

Your answer here

4

Now that we know the "best" *k*, train a new *k*-NN model with it using the entirety of the training data and evaluate it on the testing data.

In [97]:
```
#your code here
# Initialize the k-NN classifier with the optimal k value
optimal_knn = KNeighborsClassifier(n_neighbors=optimal_k)

# Train the classifier on the entire training set
optimal_knn.fit(trainFeats_norm, train_labels_binary)

# Evaluate the classifier on the testing set
testing_score = optimal_knn.score(testFeats_norm, test_labels_binary)

print(f"Testing Score (k={optimal_k}): {testing_score:.4f}")
```

```
Testing Score (k=27): 0.8833
```

Your answer here

4a

What is the testing score? (1 pts)

In [98]:
```
print(f"Testing Score (k={optimal_k}): {testing_score:.4f}")
```

```
Testing Score (k=27): 0.8833
```

Your answer here

4b

How does this score compare with the 1-NN model from question 3.3? Is it the best of the three models you trained in Section 3? (2 pts)

Comparison with 1-NN Model:

```
1-NN Model:

    Testing Score: 0.821

    The 1-NN model is highly overfit because it memorizes the training data, resulting in poor
generalization to the testing set.

27-NN Model:

    Testing Score: 0.8833

    The 27-NN model generalizes better because it averages over more neighbors, reducing
overfitting and improving performance on the testing set.
```

Is This the Best Model?

```
Comparison with Other Models:

    Logistic Regression: Typically has a simpler decision boundary and may underfit the data,
resulting in lower testing scores.

    SVM (RBF Kernel): Can achieve high performance but may require careful tuning of
hyperparameters (e.g., CC and γγ) to avoid overfitting or underfitting.

    k-NN with k=27k=27: Achieves a testing score of 0.8833, which is higher than the 1-NN
model and likely competitive with or better than the other models.

Conclusion:

    The 27-NN model appears to be the best of the three models trained in Section 3, as it
achieves the highest testing score (0.8833) and generalizes well to unseen data.
```

Why the 27-NN Model Performs Better:

```
Reduced Overfitting:

    By using k=27k=27, the model averages over more neighbors, reducing sensitivity to noise
and outliers in the training data.

Better Generalization:

    The larger kk value results in a smoother decision boundary, which improves the model's
```

ability to generalize to new data.

Optimal Trade-Off:

The 27-NN model strikes a good balance between bias and variance, avoiding both overfitting (small k) and underfitting (very large k).