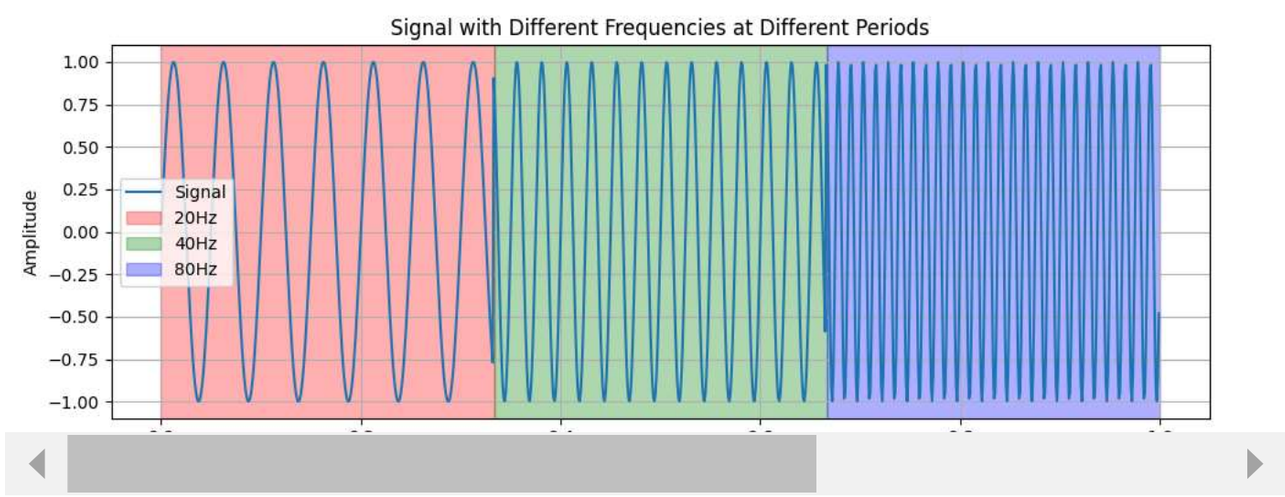
## Signal:

Any physical quantity that carries some information and varies with respect to time (then it would be time-series), or any other independent variable

```
import numpy as np
import matplotlib.pyplot as plt

# Parameters
fs = 1000  # Sampling frequency in Hz
T = 1.0    # Duration in seconds
t = np.linspace(0, T, int(T * fs), endpoint=False)  # Time vector

# Create a signal with different frequencies at different periods
f1 = 20   # Frequency of the first segment (Hz)
f2 = 40  # Frequency of the second segment (Hz)
f3 = 80  # Frequency of the third segment (Hz)

# Define different segments
segment1 = np.sin(2 * np.pi * f1 * t[:int(len(t) / 3)])
segment2 = np.sin(2 * np.pi * f2 * t[int(len(t) / 3):int(2 * len(t) / 3)])
segment3 = np.sin(2 * np.pi * f3 * t[int(2 * len(t) / 3):])

# Combine segments into one signal
signal = np.concatenate((segment1, segment2, segment3))

# Plot the signal
plt.figure(figsize=(10, 4))
plt.plot(t, signal, label='Signal')
plt.title('Signal with Different Frequencies at Different Periods')
plt.xlabel('Time [s]')
plt.ylabel('Amplitude')
plt.grid(True)

# Highlight different frequency segments
plt.axvspan(0, T/3, color='red', alpha=0.3, label='20Hz')
plt.axvspan(T/3, 2*T/3, color='green', alpha=0.3, label='40Hz')
plt.axvspan(2*T/3, T, color='blue', alpha=0.3, label='80Hz')

plt.legend()
plt.tight_layout()
plt.show()

```




## Fourier Transform

- The Fourier Transform is a mathematical operation that **transforms a signal from the time domain to the frequency domain**.
- It **breaks down a complex signal into (up to infinite sum of) sinusoidal components (Fourier Series)**, revealing the different frequencies present in the signal and their corresponding amplitudes.
- This is **useful for analyzing periodic behaviors and identifying dominant frequencies within a signal**, which are often difficult to observe directly in the time domain.

The Fourier series representation of a periodic function $f(t)$ with period $T$ is given by:

$$f(t) = \frac{a_0}{2} + \sum_{n=1}^{\infty} (a_n \cos(n\omega t) + b_n \sin(n\omega t))$$

where:

- $\omega = \frac{2\pi}{T}$ is the fundamental or angular frequency
- $n = 1, 2, 3, \ldots$ are the harmonic frequencies (integer multiples of the fundamental frequency)
- $a_0$, $a_n$, and $b_n$ are the Fourier coefficients

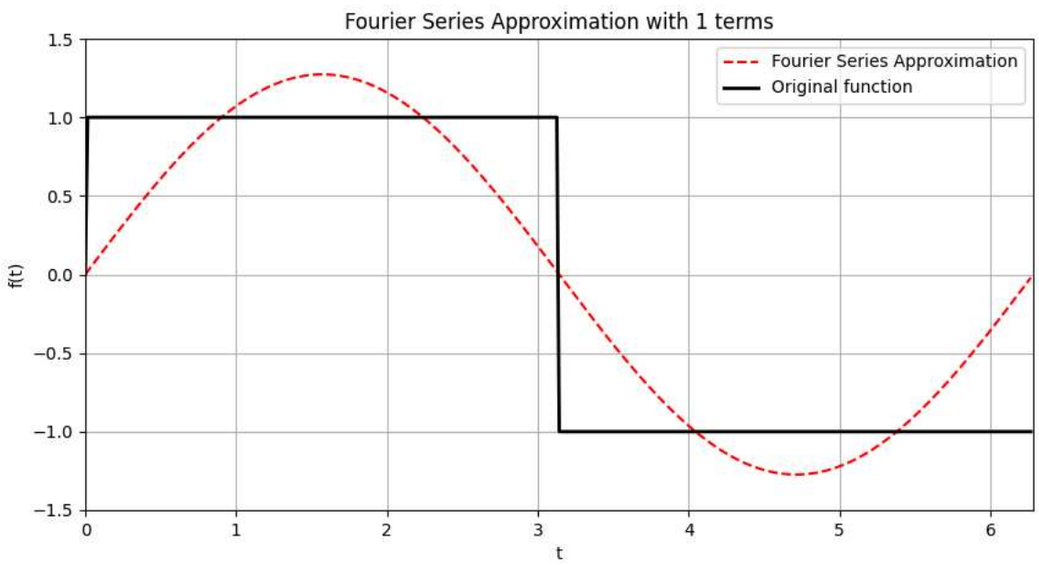
The Fourier coefficients are calculated using the following integrals:

$$a_0 = \frac{1}{T} \int_0^T f(t)\, dt$$

$$a_n = \frac{2}{T} \int_0^T f(t) \cos(n\omega t)\, dt$$

$$b_n = \frac{2}{T}\int_0^T f(t)\sin(n\omega t)\,dt$$

```
import numpy as np
from scipy.integrate import quad
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation
from IPython.display import HTML

def f(t):
    # Define your function here; example: a square wave
    return np.sign(np.sin(t))

# Parameters
T = 2 * np.pi  # period

# Calculate the coefficients
def a0(T):
    result, _ = quad(f, 0, T)
    return (1 / T) * result

def an(n, T):
    result, _ = quad(lambda t: f(t) * np.cos(n * 2 * np.pi * t / T), 0, T)
    return (2 / T) * result

def bn(n, T):
    result, _ = quad(lambda t: f(t) * np.sin(n * 2 * np.pi * t / T), 0, T)
    return (2 / T) * result

# Initialize parameters for the animation
t = np.linspace(0, T, 400, endpoint=False)
f_t = np.vectorize(f)(t)
N_max = 50  # Maximum number of terms in the Fourier series

# Create a figure and axis for the animation
fig, ax = plt.subplots(figsize=(10, 5))
line, = ax.plot([], [], label='Fourier Series Approximation', linestyle='--', color='red')
ax.plot(t, f_t, label='Original function', linewidth=2, color='black')
ax.set_xlim(0, T)
ax.set_ylim(-1.5, 1.5)
ax.set_title('Fourier Series Approximation')
ax.set_xlabel('t')
ax.set_ylabel('f(t)')
ax.grid(True)
ax.legend()

def init():
    line.set_data([], [])
    return line,

def animate(n):
    fourier_series = np.full_like(t, a0(T) / 2)  # Start with a0/2
    for k in range(1, n + 1):
        fourier_series += an(k, T) * np.cos(k * 2 * np.pi * t / T) + bn(k, T) * np.sin(k * 2 * np.pi * t / T)
    line.set_data(t, fourier_series)
    ax.set_title(f'Fourier Series Approximation with {n} terms')
    return line,

# Create the animation
ani = FuncAnimation(fig, animate, init_func=init, frames=range(1, N_max + 1), interval=200, blit=True)

# Clear the current figure before displaying the animation
plt.close(fig)

# Display the animation
HTML(ani.to_jshtml())

```



# Sampling and Sample Rate!

What is the proper sample rate to be able to capture the frequencies existing in a time-series?

The Nyquist–Shannon sampling theorem is a fundamental principle in the field of signal processing and information theory. It states that **for a continuous-time signal to be properly reconstructed from its sampled values, the sampling rate must be bigger than at least twice the highest frequency component present in the signal**.

Mathematically, the theorem can be expressed as:

$$f_s > 2f_{max}$$

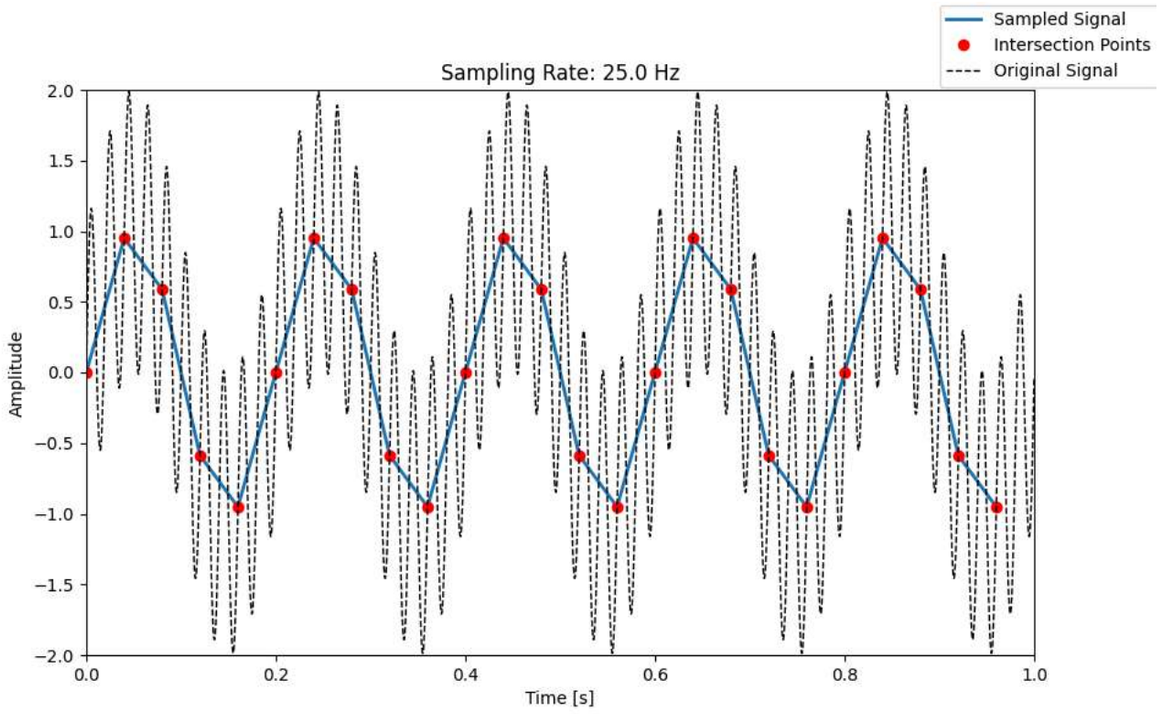
Where:

- $f_s$ is the sampling rate (samples per second)
- $f_{max}$ is the maximum frequency present in the signal
- $f_s = 2f_{max}$ is known as the Nyquist rate.

**If $f_s > 2f_{max}$ is not met, a phenomenon known as aliasing occurs.**
Aliasing is the effect where **high-frequency components in the original signal appear as lower frequencies in the sampled signal**, leading to distortion and incorrect reconstruction of the original signal.

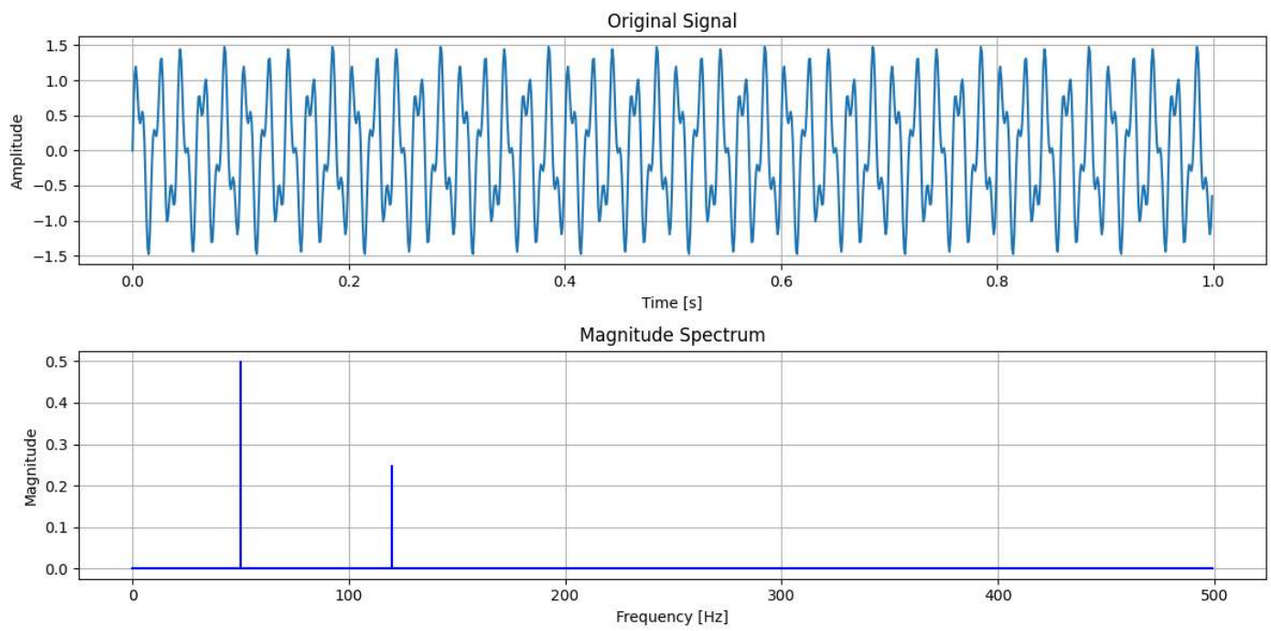
- **Question:** Why are 44100Hz and 48000Hz sampling rates quite popular for audio signals?

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation
from IPython.display import HTML

# Create a signal with two different frequencies
fs = 10000  # Original sampling frequency
t = np.linspace(0, 1, fs, endpoint=False)  # Time vector
f1 = 5  # Frequency of the first sine wave
f2 = 50  # Frequency of the second sine wave
signal = np.sin(2 * np.pi * f1 * t) + np.sin(2 * np.pi * f2 * t)

# Define sampling rates around the Nyquist rate
nyquist_rate = 2 * f2  # 2 * highest frequency
sampling_rates = [nyquist_rate / 4, nyquist_rate / 2, nyquist_rate, nyquist_rate , 1.5 * nyquist_rate, 2 * nyquist_rate, 3 * nyquist_rate

fig, ax = plt.subplots(figsize=(10, 6))
line, = ax.plot([], [], lw=2, label='Sampled Signal')
points, = ax.plot([], [], 'ro', label='Intersection Points')
ax.plot(t, signal, 'k--', lw=1, label='Original Signal')
ax.set_xlim(0, 1)
ax.set_ylim(-2, 2)
ax.set_title('Effect of Different Sampling Rates on Signal')
ax.set_xlabel('Time [s]')
ax.set_ylabel('Amplitude')

# Position legend outside the plot area
ax.legend(loc='upper left', bbox_to_anchor=(0.9, 1.15), borderaxespad=0.)

def init():
    line.set_data([], [])
    points.set_data([], [])
    return line, points

def animate(i):
    rate = sampling_rates[i]
    t_sampled = t[::int(fs/rate)]

    signal_sampled = np.sin(2 * np.pi * f1 * t_sampled) + np.sin(2 * np.pi * f2 * t_sampled)
    line.set_data(t_sampled, signal_sampled)

    # Find intersection points
    if rate <= nyquist_rate:
        t_intersections = np.intersect1d(t, t_sampled)
        signal_intersections = np.sin(2 * np.pi * f1 * t_intersections) + np.sin(2 * np.pi * f2 * t_intersections)
        points.set_data(t_intersections, signal_intersections)
    else:
        points.set_data([], [])

    ax.set_title(f'Sampling Rate: {rate} Hz')

    return line, points

ani = FuncAnimation(fig, animate, init_func=init, frames=len(sampling_rates), interval=1000, blit=True)

# Clear the current figure before displaying the animation
plt.close(fig)

# Display the animation
HTML(ani.to_jshtml())

```




## Spectral Analysis in Python

```python
import numpy as np
import matplotlib.pyplot as plt

# Parameters
fs = 1000  # Sampling frequency in Hz
T = 1.0    # Duration in seconds
t = np.linspace(0, T, int(T * fs), endpoint=False)  # Time vector

# Create a signal with two different frequencies
f1 = 50   # Frequency of the first sine wave (Hz)
f2 = 120  # Frequency of the second sine wave (Hz)
signal = np.sin(2 * np.pi * f1 * t) + 0.5 * np.sin(2 * np.pi * f2 * t)

# Perform FFT
N = len(signal)
fft_result = np.fft.fft(signal)
fft_freq = np.fft.fftfreq(N, 1/fs)

# Only take the positive frequencies and corresponding FFT results
positive_freqs = fft_freq[:N//2]
positive_fft_result = fft_result[:N//2]

# Magnitude of the FFT (normalized)
magnitude = np.abs(positive_fft_result) / N

# Plot the original signal
plt.figure(figsize=(12, 6))

plt.subplot(2, 1, 1)
plt.plot(t, signal)
plt.title('Original Signal')
plt.xlabel('Time [s]')
plt.ylabel('Amplitude')
plt.grid(True)

# Plot the FFT (magnitude spectrum)
plt.subplot(2, 1, 2)
plt.stem(positive_freqs, magnitude, 'b', markerfmt=" ", basefmt="-b")
plt.title('Magnitude Spectrum')
plt.xlabel('Frequency [Hz]')
plt.ylabel('Magnitude')
plt.grid(True)

plt.tight_layout()
plt.show()

```




## Spectral analysis in real-worlds

- Non-periodic signals
- Truncated signals

The "edge effect" in spectral analysis occurs when a signal is truncated (e.g., when using a sliding window) or non-periodic. Here's why it happens:

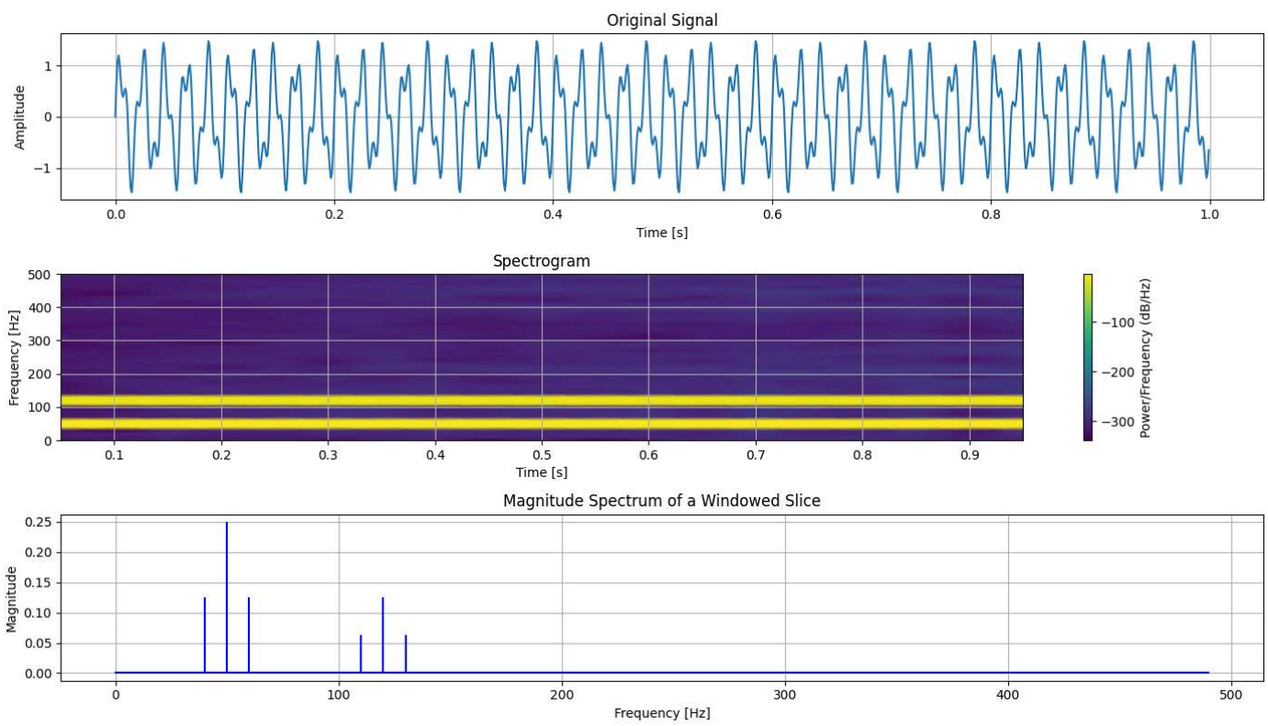
1. **Abrupt Changes at Window Edges**: When you apply a window to the time-series, the segments taken from the signal end abruptly at the window's boundaries. This truncation introduces discontinuities at the edges, which create artifacts in the frequency domain. Specifically, Fourier analysis assumes that the signal is continuous or repeats beyond the window's boundaries, or the integration boundaries contain full periods of the signal. So sudden changes result in spectral leakage.
2. **Spectral Leakage**: The abrupt start and end of the truncated signal segment act as high-frequency components, which appear as extra, unwanted frequencies in the analysis. These frequencies may interfere with the actual frequencies present in the signal, causing "leakage" across the spectrum and reducing the precision of frequency estimates.
3. **Loss of Periodicity**: Fourier analysis, especially using the Discrete Fourier Transform (DFT), assumes periodic signals within each window. If the original signal is not periodic, truncating it for each window creates an artificial period at the window boundaries. This introduces discontinuities in each period, resulting in extra frequency components that are not part of the true signal.
4. **Reduced Frequency Resolution**: Shorter windows (due to truncation) also reduce the frequency resolution, limiting the ability to accurately distinguish between closely spaced frequencies. This effect is amplified if the signal changes in each window, as it complicates distinguishing true signal frequencies from those introduced by window truncation.

### Mitigating Edge Effects (for further reading)

Using windowing functions (like Hamming or Hanning windows) can help reduce edge effects by gradually tapering the signal at the window edges, smoothing out abrupt changes. This reduces spectral leakage, although it doesn't eliminate it entirely.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.signal import windows, spectrogram

# Parameters
fs = 1000  # Sampling frequency in Hz
T = 1.0    # Duration in seconds
t = np.linspace(0, T, int(T * fs), endpoint=False)  # Time vector

# Create a signal with two different frequencies
f1 = 50   # Frequency of the first sine wave (Hz)
f2 = 120  # Frequency of the second sine wave (Hz)
signal = np.sin(2 * np.pi * f1 * t) + 0.5 * np.sin(2 * np.pi * f2 * t)

# Parameters for windowing
window_length = 100  # Length of the window
window_type = 'hann'  # Type of the window (e.g., 'hann', 'hamming', etc.)
window = windows.get_window(window_type, window_length)
n_overlap = window_length // 2  # Number of overlapping samples

# Perform STFT
frequencies, times, Sxx = spectrogram(signal, fs, window=window, nperseg=window_length, noverlap=n_overlap, scaling='spectrum')

# Plot the original signal
plt.figure(figsize=(14, 8))

plt.subplot(3, 1, 1)
plt.plot(t, signal)
plt.title('Original Signal')
plt.xlabel('Time [s]')
plt.ylabel('Amplitude')
plt.grid(True)
```

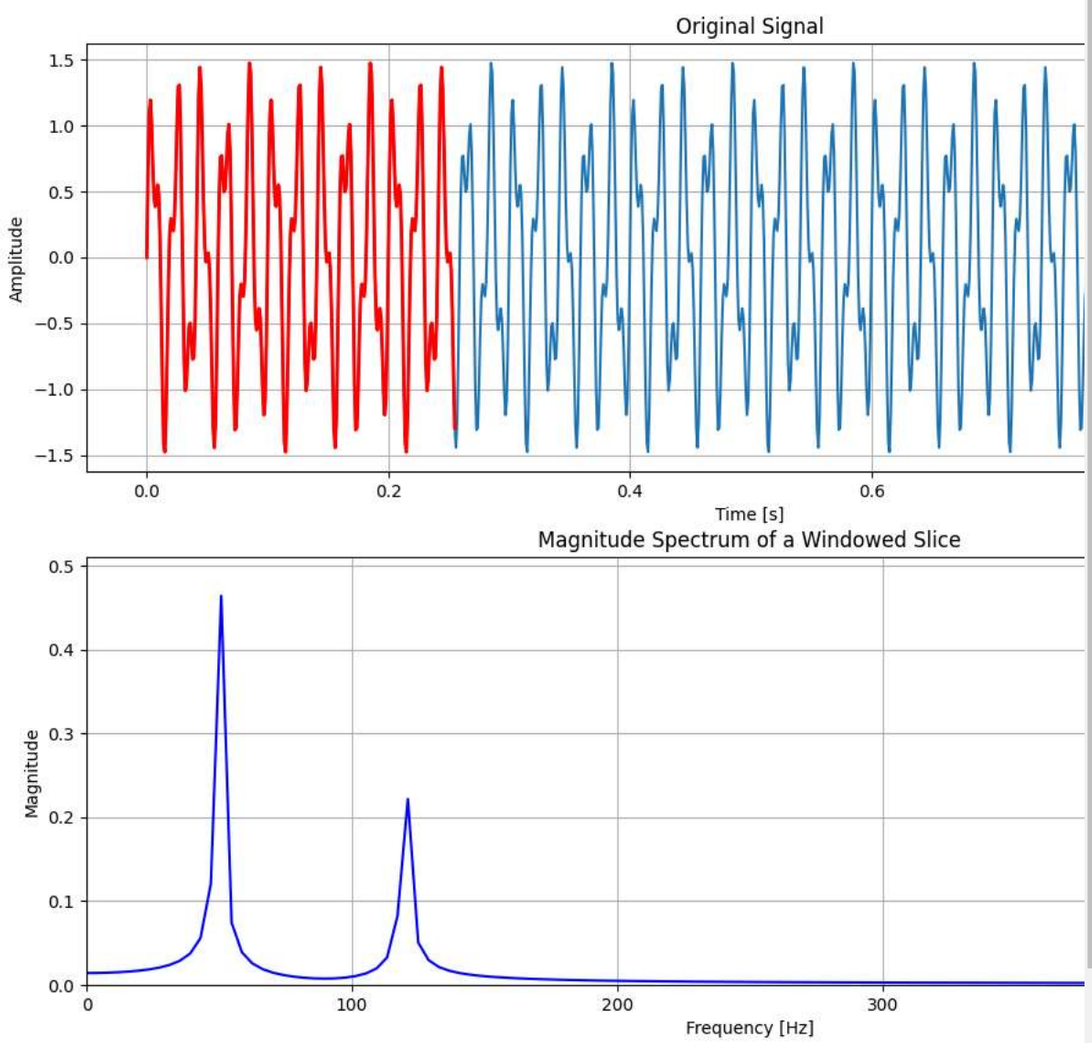
```
# Plot the spectrogram
plt.subplot(3, 1, 2)
plt.pcolormesh(times, frequencies, 10 * np.log10(Sxx), shading='gouraud')
plt.title('Spectrogram')
plt.xlabel('Time [s]')
plt.ylabel('Frequency [Hz]')
plt.colorbar(label='Power/Frequency (dB/Hz)')
plt.grid(True)

# Illustration of windowing effect
# Select a specific time slice for illustration
time_slice = int(0.5 * fs) + 110  # Center of the signal (at 0.5 seconds)
signal_slice = signal[time_slice:time_slice + window_length] * window

# Perform FFT on the windowed signal slice
N_slice = len(signal_slice)
fft_result_slice = np.fft.fft(signal_slice)
fft_freq_slice = np.fft.fftfreq(N_slice, 1/fs)

# Only take the positive frequencies and corresponding FFT results
positive_freqs_slice = fft_freq_slice[:N_slice//2]
positive_fft_result_slice = fft_result_slice[:N_slice//2]

# Magnitude of the FFT (normalized)
magnitude_slice = np.abs(positive_fft_result_slice) / N_slice

# Plot the FFT (magnitude spectrum) of the windowed signal slice
plt.subplot(3, 1, 3)
plt.stem(positive_freqs_slice, magnitude_slice, 'b', markerfmt=" ", basefmt="-b")
plt.title('Magnitude Spectrum of a Windowed Slice')
plt.xlabel('Frequency [Hz]')
plt.ylabel('Magnitude')
plt.grid(True)

plt.tight_layout()
plt.show()

```




Same story when sliding window over time-series

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation
from scipy.signal import windows
from IPython.display import HTML

# Parameters
fs = 1000  # Sampling frequency in Hz
T = 1.0    # Duration in seconds
t = np.linspace(0, T, int(T * fs), endpoint=False)  # Time vector

# Create a signal with two different frequencies
f1 = 50   # Frequency of the first sine wave (Hz)
f2 = 120  # Frequency of the second sine wave (Hz)
signal = np.sin(2 * np.pi * f1 * t) + 0.5 * np.sin(2 * np.pi * f2 * t)

# Parameters for windowing
window_length = 256  # Length of the window
window = np.ones(window_length)  # Simple rectangular window

# Setup figure and axes
fig, (ax_signal, ax_spectrum) = plt.subplots(2, 1, figsize=(14, 10))
```

```
23
24 # Plot the original signal
25 ax_signal.plot(t, signal)
26 ax_signal.set_title('Original Signal')
27 ax_signal.set_xlabel('Time [s]')
28 ax_signal.set_ylabel('Amplitude')
29 ax_signal.grid(True)
30
31 line_window, = ax_signal.plot([], [], 'r', linewidth=2)
32
33 # Setup the spectrum plot
34 ax_spectrum.set_title('Magnitude Spectrum of a Windowed Slice')
35 ax_spectrum.set_xlabel('Frequency [Hz]')
36 ax_spectrum.set_ylabel('Magnitude')
37 ax_spectrum.grid(True)
38 line_spectrum, = ax_spectrum.plot([], [], 'b')
39
40 # Define the update function for animation
41 def update(frame):
42     # Compute the windowed signal slice
43     start = frame
44     end = start + window_length
45     signal_slice = signal[start:end] * window
46
47     # Perform FFT on the windowed signal slice
48     N_slice = len(signal_slice)
49     fft_result_slice = np.fft.fft(signal_slice)
50     fft_freq_slice = np.fft.fftfreq(N_slice, 1/fs)
51
52     # Only take the positive frequencies and corresponding FFT results
53     positive_freqs_slice = fft_freq_slice[:N_slice//2]
54     positive_fft_result_slice = fft_result_slice[:N_slice//2]
55
56     # Magnitude of the FFT (normalized)
57     magnitude_slice = np.abs(positive_fft_result_slice) / N_slice
58
59     # Update the windowed signal slice plot
60     line_window.set_data(t[start:end], signal[start:end] * window)
61
62     # Update the spectrum plot
63     line_spectrum.set_data(positive_freqs_slice, magnitude_slice)
64
65     # Redraw the spectrum plot limits
66     ax_spectrum.set_xlim(0, fs / 2)
67     ax_spectrum.set_ylim(0, max(magnitude_slice) * 1.1)
68
69     return line_window, line_spectrum
70
71 # Create the animation
72 frames = range(0, len(signal) - window_length, window_length // 2)
73 ani = FuncAnimation(fig, update, frames=frames, blit=True)
74
75 # Clear the current figure before displaying the animation
76 plt.close(fig)
77
78 # Display the animation
79 HTML(ani.to_jshtml())
80
```




### Checkpoint

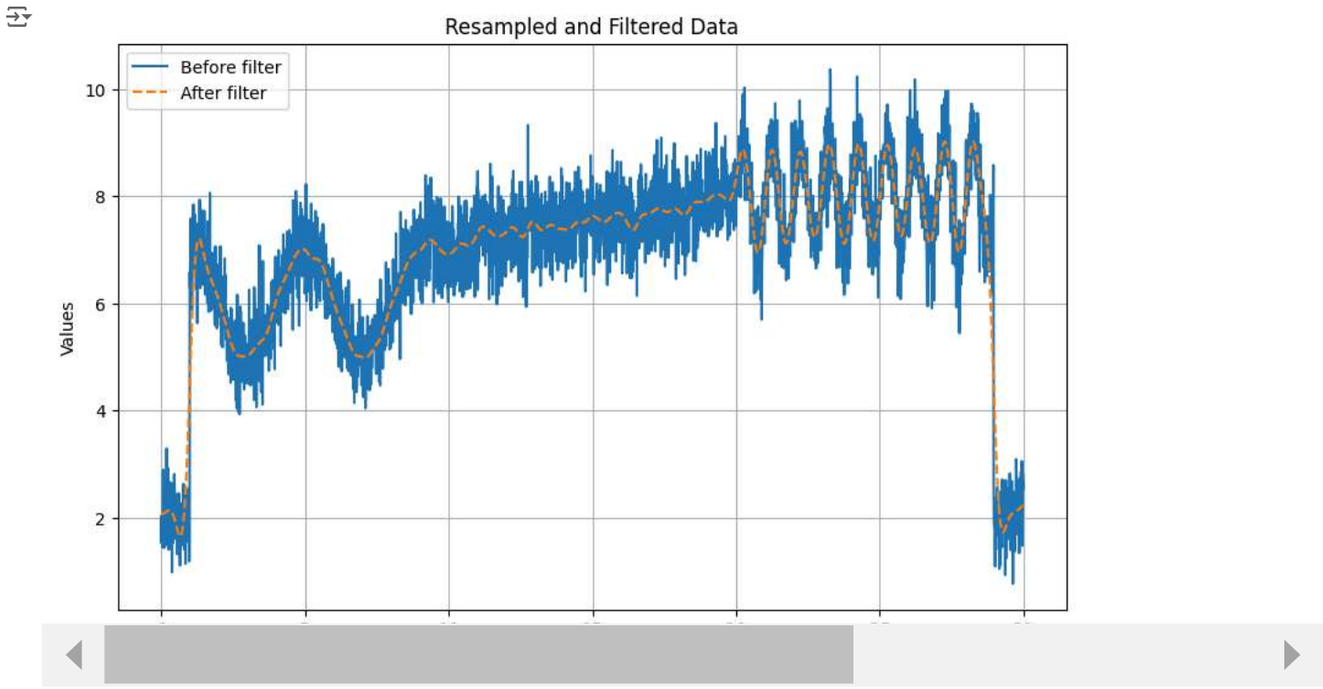
- What is the frequency of human motion?
- How should we choose the sampling rate for our motion sensors?

## Filtering

What frequencies are we interested in?

```
'''Trying different low pass filters'''

import numpy as np
from scipy.signal import butter,filtfilt
from scipy.interpolate import interp1d
import matplotlib.pyplot as plt

# Generating example data (time and values)
time_original_1 = np.arange(1, 9, 1/20)  # Original time array (20 Hz)
values_original_1 = np.sin(0.5 * np.pi * time_original_1) + 4  # Example values

time_original_2 = np.arange(20, 29, 1/20)  # Original time array (20 Hz)
values_original_2 = np.sin(2 * np.pi * time_original_2) + 6 # Example values

# Combine the two sets of example data together
combined_time = np.concatenate((time_original_1, time_original_2))
combined_values = np.concatenate((values_original_1, values_original_2))

# Define new time array with larger time interval (100 Hz)
time_resampled = np.arange(0, 30, 1/100)  # New time array (100 Hz)

# Use interp1d to interpolate and extrapolate the data
interpolator = interp1d(combined_time, combined_values, kind='slinear', fill_value=(0, 0), bounds_error=False)

# Interpolate/extrapolate the values to the new time array
values_resampled = interpolator(time_resampled) + 2 + np.random.normal(0, 0.5, len(time_resampled))


def butter_lowpass_filter(data, cutoff, fs, order):
    nyq = 0.5 * fs
    normal_cutoff = cutoff / nyq
    # Get the filter coefficients
```

```
    b, a = butter(order, normal_cutoff, btype='low', analog=False)
    y = filtfilt(b, a, data)
    return y

values_filtered = butter_lowpass_filter(values_resampled, 1.5, 100, 3)


# Plot the original and resampled data
plt.figure(figsize=(10, 6))
plt.plot(time_resampled, values_resampled, label='Before filter', marker='', linestyle='-')
plt.plot(time_resampled, values_filtered, label='After filter', marker='', linestyle='--')
plt.xlabel('Time')
plt.ylabel('Values')
plt.title('Resampled and Filtered Data')
plt.legend()
plt.grid(True)
plt.show()

```




## Feature engineering for time-series

Feature engineering for time-series data is essential for building effective models, especially when the raw data doesn't directly reveal patterns suitable for predictions or classification.

Here's a breakdown of key domains for feature engineering in time-series:

### 1. **Time Domain**

- **Description**: Time-domain features are extracted directly from the values of the time-series signal. These features help capture general characteristics, trends, or periodic behaviors in the time domain.
- **Examples**:
  - **Mean**: Average value of the series over a specific window or the entire series.
  - **Standard Deviation**: Variability of the signal within a window.
  - **Peak-to-Peak**: Difference between the maximum and minimum values in a window, indicating the range of fluctuations.
  - **Autocorrelation**: Measure of similarity between observations at different time lags, indicating repeated patterns.
  - **Example Use Case**: In monitoring stock prices, the moving average can reveal trends over time, while peak-to-peak values can reflect volatility.

### 2. **Statistical Domain**

- **Description**: Statistical features provide insights into the distribution, central tendency, and variability of the data, offering a high-level summary of the time-series.
- **Examples**:
  - **Skewness**: Indicates whether the data is asymmetrically distributed, useful in assessing if there are more extreme values on one side.
  - **Kurtosis**: Measures the "tailedness" of the distribution, identifying if extreme deviations are frequent.
  - **Percentiles**: Different quantiles (like median, upper, and lower quartiles) give insights into the data's spread.
  - **Entropy**: Quantifies the randomness or disorder within the signal.
  - **Example Use Case**: In speech signal processing, skewness and kurtosis help identify whether certain sounds or energy levels occur more often, which aids in speaker recognition.

### 3. **Amplitude Domain**

- **Description**: Amplitude-domain features deal with the signal's magnitude or energy over time, capturing peaks and intensity levels within the signal.
- **Examples**:
  - **Root Mean Square (RMS)**: Provides the energy or power in the signal, often used to detect the strength of vibrations or audio signals.

- **Max and Min Amplitude**: Tracks the extreme amplitude values, helping detect signal spikes or drops.
- **Signal Envelope**: A smoothed outline capturing the signal's general amplitude trend.
- **Example Use Case**: In EEG signal analysis, the RMS and peak amplitudes can indicate neural activities' strength and intensity over specific periods.

### 4. Spectral Domain

- **Description**: Spectral (or frequency) domain features are obtained by transforming the signal from the time domain to the frequency domain, often using the Fourier Transform. These features reveal the frequency content and are helpful for identifying periodicities and oscillations in the signal.
- **Examples**:
  - **Dominant Frequency**: The primary frequency component in the signal, which can reveal recurring patterns or cycles.
  - **Spectral Entropy**: Quantifies the distribution of energy across different frequency bands, giving insights into the signal's complexity.
  - **Spectral Centroid**: The center of mass of the spectral power, indicating where the signal's power is concentrated.
  - **Spectral Band Power**: The power in specific frequency bands (e.g., delta, theta, alpha bands in EEG).
  - **Example Use Case**: In wearable heart monitoring, dominant frequencies can indicate heart rate, while spectral band power in EEG data can signal different brain states.

### 5. Time-Frequency Domain (for your reading)

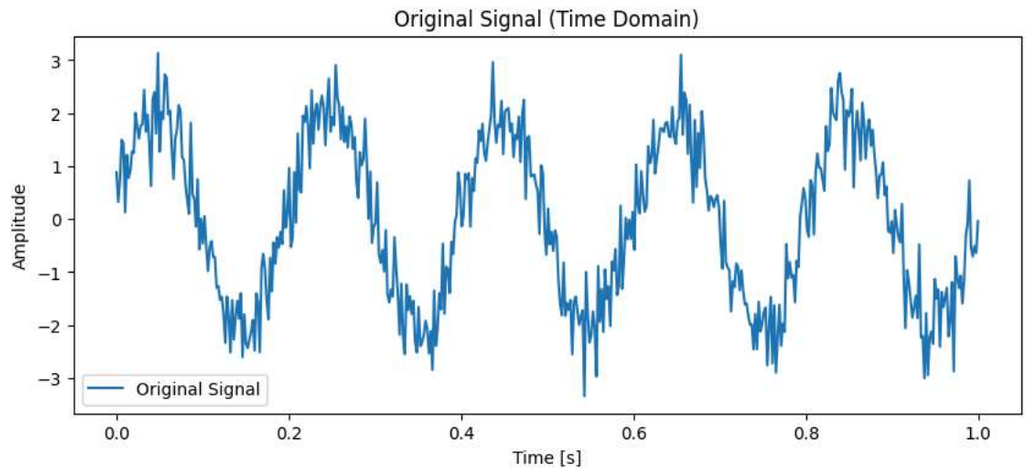
- **Description**: The time-frequency domain combines information from both time and frequency domains, capturing how frequency content changes over time. This is useful when the signal's frequency characteristics vary, as in non-stationary signals.
- **Examples**:
  - **Wavelet Transform Coefficients**: Decomposes the signal at multiple scales, capturing both frequency and time variations, commonly used for transient or abrupt changes.
  - **Short-Time Fourier Transform (STFT)**: Divides the signal into short segments and performs Fourier analysis on each, providing a spectrum for each time window.
  - **Hilbert-Huang Transform (HHT)**: An adaptive method capturing time-frequency features, suitable for non-linear and non-stationary signals.
  - **Mel-Frequency Cepstral Coefficients (MFCCs)**: Common in audio processing, MFCCs represent the signal's power spectrum in terms of Mel-frequency bands.
  - **Example Use Case**: For human activity recognition, wavelet transforms can capture quick bursts of movement, while MFCCs in speech recognition help differentiate between phonemes and tones.

Useful libraries:

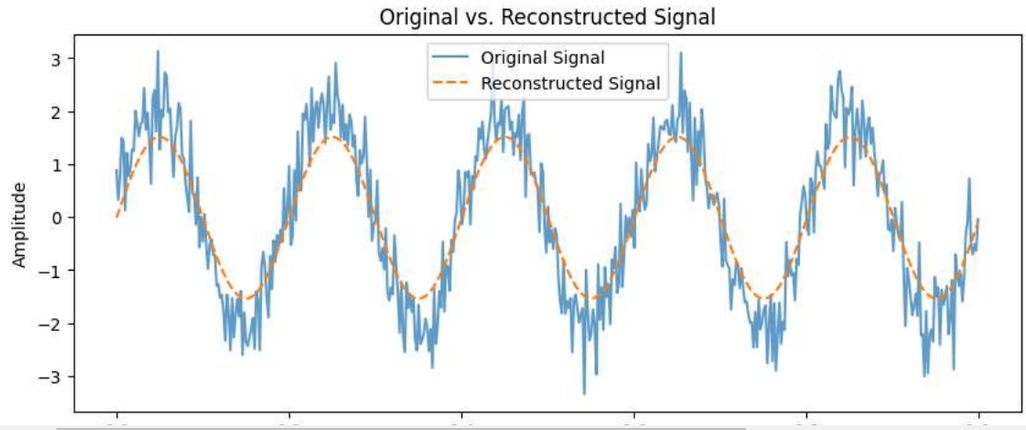
- https://tsfel.readthedocs.io/en/latest/index.html
- https://tsfresh.readthedocs.io/en/latest/

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from scipy.fft import fft, ifft, fftfreq
from scipy.stats import skew, kurtosis

# Generate an artificial time-series signal
np.random.seed(0)
time = np.linspace(0, 1, 500)  # 500 time points over 1 second
frequency = 5  # 5 Hz signal frequency
amplitude = 2
noise = np.random.normal(0, 0.5, time.shape)  # Gaussian noise
signal = amplitude * np.sin(2 * np.pi * frequency * time) + noise

# Plot the original signal
plt.figure(figsize=(10, 4))
plt.plot(time, signal, label='Original Signal')
plt.xlabel("Time [s]")
plt.ylabel("Amplitude")
plt.title("Original Signal (Time Domain)")
plt.legend()
plt.show()

# Feature Generation
# 1. Time Domain Features
peak_to_peak = np.ptp(signal)  # Difference between max and min

# 2. Statistical Features
mean_val = np.mean(signal)
std_dev = np.std(signal)
signal_skewness = skew(signal)
signal_kurtosis = kurtosis(signal)

# 3. Amplitude Domain Features
max_amplitude = np.max(signal)
min_amplitude = np.min(signal)
rms_amplitude = np.sqrt(np.mean(signal**2))  # Root mean square amplitude

# 4. Spectral Domain Features
# Perform FFT to get frequency components
signal_fft = fft(signal)
signal_freq = fftfreq(len(signal), time[1] - time[0])
power_spectrum = np.abs(signal_fft)**2

# Dominant frequency (excluding zero-frequency component)
dominant_freq = signal_freq[np.argmax(power_spectrum[1:]) + 1]
dominant_power = np.max(power_spectrum[1:])

# Display extracted features
features = {
    "Mean": mean_val,
```

```
    "Standard Deviation": std_dev,
    "Peak-to-Peak": peak_to_peak,
    "Skewness": signal_skewness,
    "Kurtosis": signal_kurtosis,
    "Max Amplitude": max_amplitude,
    "Min Amplitude": min_amplitude,
    "RMS Amplitude": rms_amplitude,
    "Dominant Frequency": dominant_freq,
    "Dominant Power": dominant_power,
}

# Converting to DataFrame
features_df = pd.DataFrame(list(features.items()), columns=["Feature", "Value"])
print("Extracted Features:")
print(features_df)

# Reconstructing the signal using key features
reconstructed_signal = rms_amplitude * np.sin(2 * np.pi * dominant_freq * time) + mean_val

# Plot the reconstructed signal
plt.figure(figsize=(10, 4))
plt.plot(time, signal, label="Original Signal", alpha=0.7)
plt.plot(time, reconstructed_signal, label="Reconstructed Signal", linestyle="--")
plt.xlabel("Time [s]")
plt.ylabel("Amplitude")
plt.title("Original vs. Reconstructed Signal")
plt.legend()
plt.show()

```




```
Extracted Features:
              Feature          Value
0                Mean      -0.012677
1  Standard Deviation       1.524065
2        Peak-to-Peak       6.470605
3            Skewness      -0.038490
4            Kurtosis      -1.193419
5        Max Amplitude       3.131301
6        Min Amplitude      -3.339304
7        RMS Amplitude       1.524118
8  Dominant Frequency       4.990000
9      Dominant Power  260240.584007
```




## Understanding the Curse of Dimensionality

- **What is the Curse of Dimensionality?**
  - The curse of dimensionality refers to the various challenges that arise when working with high-dimensional data. As the number of features (dimensions) in a dataset increases, **each data point becomes increasingly sparse in the feature space**, making it harder to identify meaningful patterns.
  - **Example**: In image recognition, a single image can contain thousands of pixels as features. While some pixels contribute to identifying objects, many do not, adding complexity and potentially introducing noise rather than useful information.
- **Impact on Models**:
  - **Distance-Based Models (e.g., k-NN, k-Means)**:

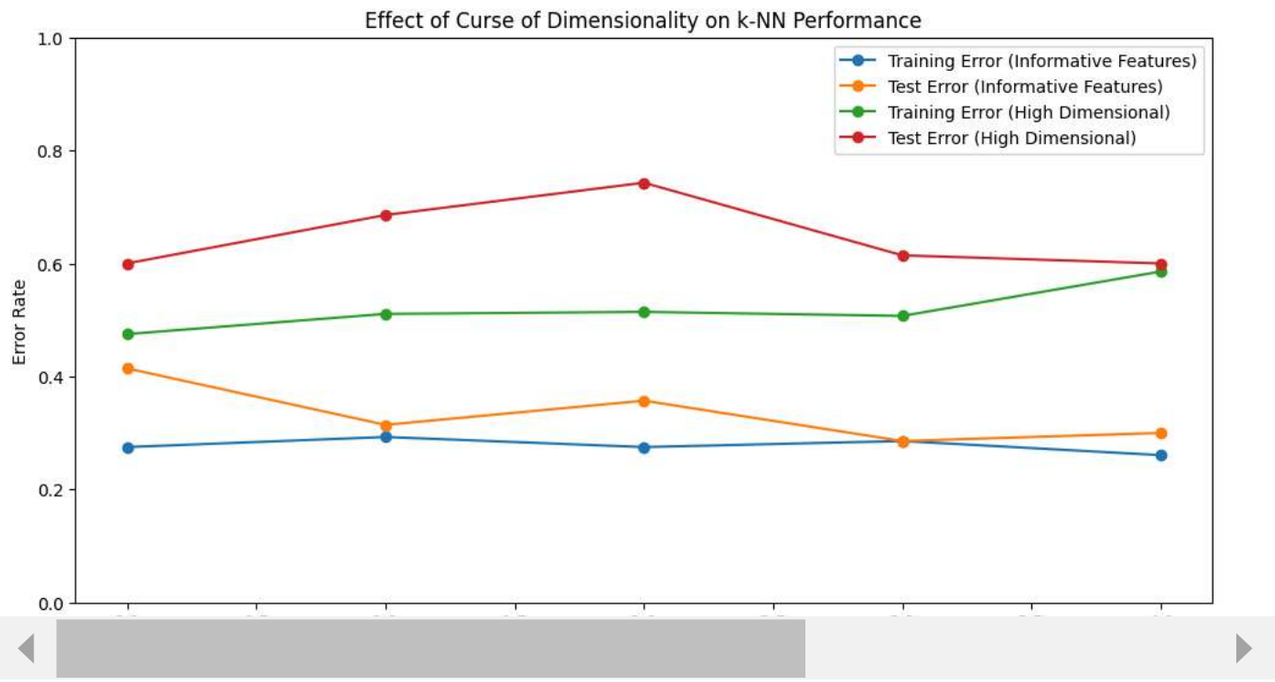
        - Distance-based models are highly affected by high dimensionality because as dimensions increase, the distances between points become less distinguishable. In k-NN, for example, as the number of irrelevant features increases, it becomes harder to find truly "nearest" neighbors, as all points end up roughly equidistant in high-dimensional space. This results in poor model accuracy, particularly on sparse datasets.
    - **Linear and Logistic Regression**:
        - High-dimensional data often **leads to overfitting** in models like linear or logistic regression, where each additional feature introduces a new parameter to learn. If irrelevant or noisy features are included, the model may "learn" from noise, resulting in poor generalization on new data.
    - **Decision Trees and Ensemble Models (e.g., Random Forest, XGBoost)**:
        - Decision trees and ensemble models tend to perform well on high-dimensional data, but their performance decreases when many irrelevant features are present. Each split in a decision tree depends on feature values; if features are mostly irrelevant, splits become noisy and add unnecessary complexity, leading to overfitting.
    - **Neural Networks**:
        - Neural networks, especially deep networks, can theoretically handle high-dimensional data by learning hierarchical representations. However, if the data includes a large number of irrelevant features, **training becomes slower**, and the network may converge to **suboptimal solutions** or require more epochs and **larger training datasets** to distinguish signal from noise effectively.
- **Computational Cost and Training Time**:
    - High-dimensional data significantly **increases computational costs** and **training time** because **each added feature increases the complexity of the model**. Training a model with many features involves more parameters, more calculations per iteration, and a greater memory requirement. This is especially problematic for large datasets, where high dimensionality can make training prohibitively slow, particularly for complex models like neural networks and ensemble models.
    - **Storage and Memory**: High-dimensional datasets also **require more storage and memory** resources for data loading and preprocessing, as each feature needs to be loaded and processed in each batch, which can slow down training and require specialized hardware in extreme cases.
- **Theoretical Solutions - More Data vs. Feature Selection**:
    - In theory, if a model has access to a much larger number of training examples, it can learn to ignore noise in high-dimensional data and focus on relevant patterns. For instance, in neural networks, with enough data, irrelevant features can be ignored over time. However, this solution has limitations:
        - **Training Cost**: Collecting and using more data significantly increases the training time and computational cost, requiring more powerful hardware and extended training sessions.
        - **Data Quality**: More data may not always be feasible or effective if the additional samples do not represent meaningful patterns or are difficult to acquire in certain domains.
    - **Feature Selection as a Practical Solution**: Instead of expanding the dataset, feature selection techniques provide a more practical approach by identifying and keeping only the most informative features, reducing the dimensionality and allowing the model to focus on the most relevant aspects of the data. This leads to improved model interpretability, lower computational costs, and faster training times.

## How Feature Dimensionality Impacts Model Performance

- **Balancing the Right Features**
    - **Underfitting**: Caused by too few features, leading to missing essential patterns.
    - **Overfitting**: Caused by too many features, causing noise to be learned as patterns.

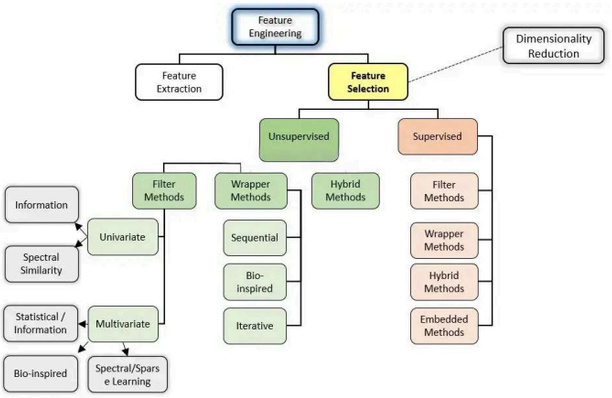
```
import numpy as np
import pandas as pd
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split, GridSearchCV, cross_val_score
from sklearn.model_selection import StratifiedKFold
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score
import matplotlib.pyplot as plt

# Generate a synthetic dataset with 5 informative features and no redundant or irrelevant features initially
X_informative, y = make_classification(
    n_samples=500, n_features=5, n_informative=4, n_redundant=0, n_classes=8, n_clusters_per_class=1, random_state=0
)

# Split the informative dataset into training and test sets
X_train_inf, X_test_inf, y_train, y_test = train_test_split(X_informative, y, test_size=0.3, random_state=42)

# Now add irrelevant features to create a high-dimensional dataset
X_high_dim = np.hstack([X_informative, np.random.normal(0, 1, (X_informative.shape[0], 100))])
X_train_high, X_test_high, _, _ = train_test_split(X_high_dim, y, test_size=0.3, random_state=42)

# Define k-NN and hyperparameter tuning for both datasets
knn = KNeighborsClassifier()
param_grid = {'n_neighbors': np.arange(1, 31)}

# Nested cross-validation for the informative dataset
cv_outer = StratifiedKFold(n_splits=5, shuffle=True, random_state=0)
train_errors_inf, test_errors_inf = [], []

for train_idx, test_idx in cv_outer.split(X_train_inf, y_train):
    X_train_outer, X_test_outer = X_train_inf[train_idx], X_train_inf[test_idx]
    y_train_outer, y_test_outer = y_train[train_idx], y_train[test_idx]

    # Inner cross-validation for hyperparameter tuning
    cv_inner = StratifiedKFold(n_splits=3, shuffle=True, random_state=0)
    grid_search = GridSearchCV(knn, param_grid, scoring='f1_weighted', cv=cv_inner)
    grid_search.fit(X_train_outer, y_train_outer)
```

```
    best_k = grid_search.best_params_['n_neighbors']

    # Evaluate with best hyperparameters
    knn_best = KNeighborsClassifier(n_neighbors=best_k)
    knn_best.fit(X_train_outer, y_train_outer)
    train_errors_inf.append(1 - knn_best.score(X_train_outer, y_train_outer))
    test_errors_inf.append(1 - knn_best.score(X_test_outer, y_test_outer))

# Nested cross-validation for the high-dimensional dataset
train_errors_high, test_errors_high = [], []

for train_idx, test_idx in cv_outer.split(X_train_high, y_train):
    X_train_outer, X_test_outer = X_train_high[train_idx], X_train_high[test_idx]
    y_train_outer, y_test_outer = y_train[train_idx], y_train[test_idx]

    # Inner cross-validation for hyperparameter tuning
    grid_search = GridSearchCV(knn, param_grid, scoring='f1_weighted', cv=cv_inner)
    grid_search.fit(X_train_outer, y_train_outer)
    best_k = grid_search.best_params_['n_neighbors']

    # Evaluate with best hyperparameters
    knn_best = KNeighborsClassifier(n_neighbors=best_k)
    knn_best.fit(X_train_outer, y_train_outer)
    train_errors_high.append(1 - knn_best.score(X_train_outer, y_train_outer))
    test_errors_high.append(1 - knn_best.score(X_test_outer, y_test_outer))

# Plotting the training and test errors for both datasets
plt.figure(figsize=(12, 6))
plt.plot(train_errors_inf, label="Training Error (Informative Features)", marker='o')
plt.plot(test_errors_inf, label="Test Error (Informative Features)", marker='o')
plt.plot(train_errors_high, label="Training Error (High Dimensional)", marker='o')
plt.plot(test_errors_high, label="Test Error (High Dimensional)", marker='o')
plt.xlabel("Cross-validation Fold")
plt.ylabel("Error Rate")
plt.ylim(0, 1)  # Set y-axis range from 0 to 1
plt.title("Effect of Curse of Dimensionality on k-NN Performance")
plt.legend()
plt.show()

```




## Types of Feature Selection Algorithms

- **Overview**:
  - **Filter Methods**: Select features based on statistical/mathematical metrics, independent of models.
  - **Wrapper Methods**: Select based on model performance.
  - **Embedded Methods**: Feature selection happens during model training.
- **Choosing the Right Method**: Each method has advantages based on dataset size, computational resources, and model needs.




source: https://medium.com/analytics-vidhya/feature-selection-extended-overview-b58f1d524c1c

## Filter Methods

- **Overview**:
  - Independent statistical/mathematical evaluation of each feature for relevance.
- **Pros and Cons**:
  - **Pros**: Computationally efficient, model-agnostic.
  - **Cons**: Usually does not account for feature interactions.
- **When to Use**: Good for initial reduction on large datasets.
- **Common Algorithms**: Here's a more detailed list of filter-based feature selection methods, including mathematical formulations and Python libraries commonly used to implement them.

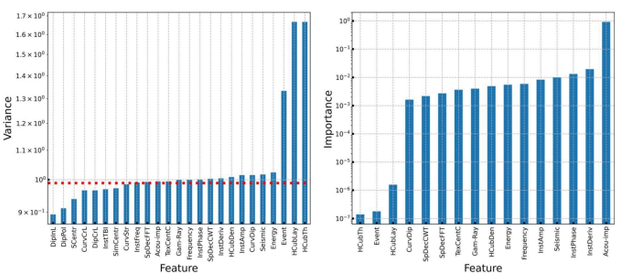
---

### 1. Variance Thresholding

- **Formulation**: Variance thresholding removes features with variance below a given threshold:

$$\text{Var}(X_j) < \text{threshold}$$

  where $X_j$ is the feature and $\text{Var}(X_j)$ is its variance.
- **Python Library**: `sklearn.feature_selection.VarianceThreshold`




source: https://doi.org/10.1029/2023EA003101

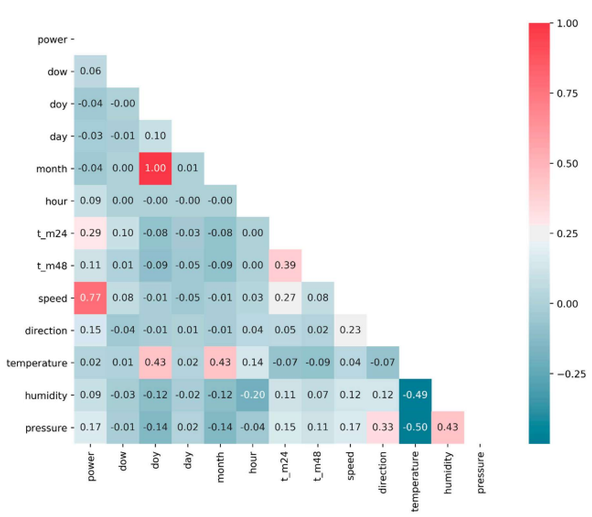
---

### 2. Correlation Analysis

- **Formulation**: Features are removed if their correlation coefficient exceeds a threshold. For linear relationships, the Pearson correlation is:

$$\rho_{X,Y} = \frac{\text{Cov}(X,Y)}{\sigma_X \sigma_Y}$$

  where $\text{Cov}(X,Y)$ is the covariance between $X$ and $Y$, and $\sigma$ represents standard deviation.
- **Python Library**: `pandas.DataFrame.corr` or `scipy.stats.pearsonr`



Source:
https://www.researchgate.net/publication/334711620_A_XGBoost_Model_with_Weather_Similarity_Analysis_and_Feature_Engineering_for_Short-Term_Wind_Power_Forecasting/figures?lo=1

### 3. Chi-Square Test

- **Formulation**: Measures the association between categorical features and target using the chi-square statistic:
- **Python Library**: `sklearn.feature_selection.chi2`

---

### 4. ANOVA (Analysis of Variance)

- **Formulation**: Analyzes variance across feature values by calculating the F-statistic:
$$F = \frac{\text{Between-group variance}}{\text{Within-group variance}}$$
where a high F-value indicates significant variance between classes.
- **Python Library**: `sklearn.feature_selection.f_classif`

---

### 5. Mutual Information (MI)

- **Formulation**: Measures dependency between feature $X$ and target $Y$ (Mutual Information is a concept from information theory that measures how much knowing one variable tells you about another. For feature selection, mutual information helps us understand how much a feature tells us about the target variable (what we're trying to predict).):
$$I(X;Y) = \sum_{x \in X} \sum_{y \in Y} p(x,y) \log \frac{p(x,y)}{p(x)p(y)}$$
where $p(x,y)$ is the joint probability, and $p(x)$ and $p(y)$ are the marginal probabilities.
- **Python Library**: `sklearn.feature_selection.mutual_info_classif`

---

### 6. ReliefF Algorithm

Nearest Neighbors: For each data point, ReliefF **finds its nearest neighbors**:

One neighbor from the same class (called the "**near hit**"). One neighbor from a different class (called the "**near miss**"). Feature Scoring:

ReliefF compares the values of each feature for the data point and its neighbors. If a feature has similar values between the data point and its "near hit" but different values between the data point and its "near miss," it's likely a useful feature. The feature's importance score increases if it consistently helps distinguish between classes. Repeat: This process is repeated for many points in the dataset, and the final score for each feature reflects how well it consistently helps separate classes across the data.

- **Formulation**: Assigns weights to features based on nearest neighbor distances. The weight for feature $X_j$ is adjusted as:
$$W_j = W_j - \frac{\text{dist}(X_j^{\text{near hit}}, X_j)}{m} + \frac{\text{dist}(X_j^{\text{near miss}}, X_j)}{m}$$
where $\text{dist}$ represents distance, $m$ is the number of samples, and "near hit" and "near miss" refer to closest samples of the same and different classes, respectively.
- **Python Library**: `skrebate` (Relief-based feature selection)

---

### 7. Information Gain

- **Formulation**: Measures the decrease in entropy from knowing feature $X$ for predicting $Y$:
$$IG(Y, X) = H(Y) - H(Y|X)$$
where $H(Y)$ is the entropy of $Y$ and $H(Y|X)$ is the conditional entropy of $Y$ given $X$.
- **Python Library**: `sklearn.feature_selection.mutual_info_classif` (can calculate information gain by computing entropy reduction)

---

### 8. Minimum Redundancy Maximum Relevance (mRMR)

- **Formulation**: Balances feature relevance (correlation with target) and redundancy (correlation among features):
$$\text{mRMR} = \max \left( \text{Relevance}(X_j, Y) - \frac{1}{|S|} \sum_{X_k \in S} \text{Redundancy}(X_j, X_k) \right)$$
where $S$ is the set of selected features.
- **Python Library**: `pymrmr` (Python package for mRMR)

## Wrapper Methods

- **Overview**:
  - Wrapper methods evaluate subsets of features based on how they impact model performance, often by using a model's prediction performance as a selection criterion. This iterative process identifies the subset of features that maximizes model performance, accounting for interactions between features.
  - These methods can lead to highly optimized models but are generally computationally intensive because they require repeated training and evaluation of the model.
- **Pros and Cons**:
  - **Pros**: Accounts for feature interactions, often providing highly accurate feature subsets for model training.
  - **Cons**: Computationally expensive, especially with large datasets or complex models, due to repeated training across multiple feature subsets. Also they are optimized for the model under study and not other models.
- **When to Use**: Wrapper methods are especially suitable for smaller datasets and cases where high model performance is critical. They are also useful when interactions between features might play a significant role in model accuracy.

---

## Common Algorithms

### 1. Sequential Feature Selection (SFS)

SFS is a greedy search algorithm that builds an optimal feature subset sequentially, either by adding features (forward selection) or removing features (backward elimination), and stopping based on model performance.

- **Forward Selection**:
- **Description**: Starts with an empty feature set and adds one feature at a time that most improves model performance, continuing until adding additional features does not improve the model.
- **Formulation**:

$$\text{Forward Step: } S_{k+1} = S_k \cup \{\arg\max_{j \notin S_k} \text{Perf}(S_k \cup \{j\})\}$$

where $S_k$ is the current feature subset, and `Perf` is a performance metric (like accuracy or F1-score).

- **Pros**: Less computationally demanding than exhaustive methods; good for finding feature subsets in moderate-sized datasets.
- **Cons**: Can miss optimal feature combinations due to the greedy approach.
- **Python Library**: `mlxtend.feature_selection.SequentialFeatureSelector`
- **Backward Elimination**:
- **Description**: Starts with the full feature set, removes the least significant feature one at a time, and re-evaluates until removing additional features reduces performance.
- **Formulation**:

$$\text{Optimal Subset} = \arg\max_{S \subseteq \{1,\ldots,n\}} \text{Perf}(S)$$

where $S_k$ is the current feature subset, and `Perf` represents the model performance metric.

- **Pros**: Useful when starting with a larger feature set and aiming to reduce dimensionality.
- **Cons**: More computationally intensive than forward selection, especially with high-dimensional datasets.
- **Python Library**: `mlxtend.feature_selection.SequentialFeatureSelector`

---

### 2. Exhaustive Feature Selection

- **Description**: Evaluates all possible feature combinations to find the subset that provides the best model performance. Since it evaluates every subset, it guarantees finding the optimal feature set for a given model.
- **Formulation**: For $n$ features, the exhaustive approach evaluates each subset $S$ of the power set $2^n$:

$$\text{Optimal Subset} = \arg\max_{S \subseteq \{1,\ldots,n\}} \text{Perf}(S)$$

where `Perf` is a performance metric like accuracy, F1-score, or AUC.

- **Pros**: Provides the best possible feature subset.
- **Cons**: Extremely computationally expensive and impractical for large datasets; limited to small feature sets due to the exponential growth in combinations.
- **Python Library**: `mlxtend.feature_selection.ExhaustiveFeatureSelector`

---

- **Real-World Example**: In healthcare prediction models, where the importance of specific features (such as test results) is critical, Recursive Feature Elimination (RFE) is often used to select the most predictive medical test results for diagnostics. For example, in cancer diagnostics, RFE might identify a small number of biomarkers from a large set, improving model interpretability and focusing on the most relevant indicators.

# Embedded Methods

- **Overview**:
  - Embedded methods select features as part of the model training process. These methods incorporate regularization or feature importance scoring within the model itself, thereby eliminating the need for an external feature selection step.
- **Pros and Cons**:
  - **Pros**: Embedded methods balance performance and computational efficiency since feature selection is integrated into the training process, and they can account for feature interactions.
  - **Cons**: They are often model-specific and rely on selection mechanisms inherent to the chosen algorithm, meaning they may not be transferrable across different models.
- **When to Use**: Embedded methods are ideal when using models with built-in feature selection capabilities, such as linear models with regularization, decision trees, and ensemble methods.

---

## Common Algorithms

### 1. LASSO (L1 Regularization)

- **Description**: LASSO (Least Absolute Shrinkage and Selection Operator) adds an $L1$-norm penalty to the model's cost function. This forces some feature coefficients to become exactly zero, effectively removing them from the model and resulting in a sparse model with only the most important features.
- **Formulation**: For a linear regression model with parameters $\beta$, the LASSO objective function is:

$$\text{Minimize} \quad \frac{1}{2n} \sum_{i=1}^{n} \left( y_i - \beta_0 - \sum_{j=1}^{p} X_{ij}\beta_j \right)^2 + \lambda \sum_{j=1}^{p} |\beta_j|$$

where $\lambda$ is the regularization parameter controlling the strength of feature selection.

- **Pros**: Effective in high-dimensional data, performs feature selection and regularization simultaneously.
- **Cons**: May select only one feature from a set of highly correlated features.
- **Python Library**: `sklearn.linear_model.Lasso`

---

### 2. Ridge Regression (L2 Regularization)

- **Description**: Ridge regression adds an $L2$-norm penalty to the model, shrinking coefficients but not setting them to zero, making it useful for identifying smaller yet still influential features.
- **Formulation**: The Ridge regression objective function is:

$$\text{Minimize} \quad \frac{1}{2n}\sum_{i=1}^{n}\left(y_i - \beta_0 - \sum_{j=1}^{p} X_{ij}\beta_j\right)^2 + \lambda\sum_{j=1}^{p}\beta_j^2$$

  where $\lambda$ controls the strength of the penalty.
- **Pros**: Helps prevent overfitting; retains all features, which can be advantageous when removing features entirely is undesirable.
- **Cons**: Does not create a sparse model (i.e., does not fully eliminate features).
- **Python Library**: `sklearn.linear_model.Ridge`

---

### 3. Elastic Net Regularization

- **Description**: Elastic Net combines both $L1$ and $L2$ regularization to retain the benefits of LASSO (sparsity) and Ridge regression (stability with correlated features). It is suitable for cases with highly correlated features.
- **Formulation**: For parameters $\beta$, the Elastic Net objective function is:

$$\text{Minimize} \quad \frac{1}{2n}\sum_{i=1}^{n}\left(y_i - \beta_0 - \sum_{j=1}^{p} X_{ij}\beta_j\right)^2 + \lambda_1\sum_{j=1}^{p}|\beta_j| + \lambda_2\sum_{j=1}^{p}\beta_j^2$$

  where $\lambda_1$ and $\lambda_2$ control the $L1$ and $L2$ penalties, respectively.
- **Pros**: Can handle correlated features and offers flexible regularization.
- **Cons**: Requires tuning two regularization parameters.
- **Python Library**: `sklearn.linear_model.ElasticNet`

---

### 4. Tree-Based Feature Importance

**Impurity** in decision trees is **a measure of how mixed the classes are within a node (or group) of the tree**. For **feature selection**, impurity helps to decide which features are most helpful in separating the data into distinct classes. Decision trees use this measure to choose the best features step-by-step as the tree grows.

1. **Impurity Measures**:
   - **Gini Impurity** and **Entropy** are common impurity measures. They assess how "pure" or "impure" a group is:
     - **Pure**: If a node has instances from only one class, it's "pure," meaning it has zero impurity.
     - **Impure**: If a node has instances from multiple classes, it's "impure" (higher impurity score).
2. **Feature Selection by Reducing Impurity**:
   - The tree tests each feature to see how much it can **reduce impurity** by splitting the data.
   - A feature that reduces impurity the most is chosen as the next split in the tree, making it an important feature.
3. **Feature Importance Scores**:
   - As the tree grows, it calculates how much each feature reduces impurity across all splits.
   - Features that consistently reduce impurity by a lot get high importance scores, making them strong predictors.

**Why It's Useful**:

- **Higher scores** mean a feature is more useful for classification, while **lower scores** indicate that the feature doesn't help much and might be dropped.

  For example, in a tree predicting loan approval, "income" might have a high importance score if it consistently helps separate approved from non-approved cases, while "zip code" might have a lower score if it doesn't reduce impurity much.

  - **Pros**: Intuitive and useful for both classification and regression tasks; accounts for interactions between features.
  - **Cons**: Biased toward features with more levels (in categorical data).
  - **Python Library**: `sklearn.ensemble.RandomForestClassifier` or `DecisionTreeClassifier`

---

### 5. Gradient Boosting Feature Importance

- **Description**: Gradient boosting models rank features by the frequency and effectiveness of their splits in each decision tree, thereby providing a feature importance score based on how often and effectively a feature is used across trees.
- **Pros**: Effective for complex datasets and provides highly accurate models.
- **Cons**: Computationally intensive due to iterative training of multiple trees.
- **Python Library**: `sklearn.ensemble.GradientBoostingClassifier`

---

### 6. Penalized Logistic Regression

- **Description**: Logistic regression with an $L1$ or $L2$ penalty can be used to select features. The penalty term shrinks less important features, with $L1$ setting some coefficients exactly to zero for sparsity.
- **Formulation**: The objective function with $L1$ regularization is:

$$\text{Minimize} \quad -\sum_{i=1}^{n} (y_i \log(p_i) + (1 - y_i) \log(1 - p_i)) + \lambda \sum_{j=1}^{p} |\beta_j|$$

where $p_i$ is the predicted probability and $\lambda$ controls the penalty.

- **Pros**: Effective for binary classification tasks, especially when many features are uninformative.
- **Cons**: Sensitive to multicollinearity; requires careful parameter tuning.

## Comparing Feature Selection Methods

- **Performance and Efficiency**:
  - **Filter Methods**: Fastest; use as a first-pass technique for dimensionality reduction.
  - **Wrapper Methods**: Best for capturing feature interactions but computationally intensive.
  - **Embedded Methods**: Balanced; efficient for models with built-in selection.
- **Choosing the Right Method**:
  - **Filter Methods**: When dealing with large datasets and quick feature selection is needed.
  - **Wrapper Methods**: For moderate datasets where accuracy is critical.
  - **Embedded Methods**: When using models that integrate feature selection.

## Conclusion

- **Key Takeaways**:
  - Effective feature engineering and selection are essential for optimizing model performance.
  - Consider the curse of dimensionality, balancing too few or too many features.
  - Select methods based on dataset size, computational resources, and model type.
  - Remember other considerations like feature stability, transformations, and encoding choices.

```
import numpy as np
import pandas as pd
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split, GridSearchCV, StratifiedKFold
from sklearn.metrics import accuracy_score
import matplotlib.pyplot as plt
from xgboost import XGBClassifier

# Generate a synthetic dataset with 5 informative features and no redundant or irrelevant features initially
X_informative, y = make_classification(
    n_samples=500, n_features=5, n_informative=4, n_redundant=0, n_classes=8, n_clusters_per_class=1, random_state=0
)

# Split the informative dataset into training and test sets
X_train_inf, X_test_inf, y_train, y_test = train_test_split(X_informative, y, test_size=0.3, random_state=42)

# Now add irrelevant features to create a high-dimensional dataset
X_high_dim = np.hstack([X_informative, np.random.normal(0, 1, (X_informative.shape[0], 100))])
X_train_high, X_test_high, _, _ = train_test_split(X_high_dim, y, test_size=0.3, random_state=42)

# Define XGBoost classifier and hyperparameter grid
xgb = XGBClassifier(eval_metric='mlogloss')
param_grid = {
    'max_depth': [5, 7],
    'learning_rate': [0.1, 0.2],
    'n_estimators': [50, 100]
}

# Nested cross-validation for the informative dataset
cv_outer = StratifiedKFold(n_splits=5, shuffle=True, random_state=0)
train_errors_inf, test_errors_inf = [], []

for train_idx, test_idx in cv_outer.split(X_train_inf, y_train):
    X_train_outer, X_test_outer = X_train_inf[train_idx], X_train_inf[test_idx]
    y_train_outer, y_test_outer = y_train[train_idx], y_train[test_idx]

    # Inner cross-validation for hyperparameter tuning
    cv_inner = StratifiedKFold(n_splits=3, shuffle=True, random_state=0)
    grid_search = GridSearchCV(xgb, param_grid, scoring='f1_weighted', cv=cv_inner, n_jobs=-1)
    grid_search.fit(X_train_outer, y_train_outer)
    best_params = grid_search.best_params_

    # Evaluate with best hyperparameters
    xgb_best = XGBClassifier(**best_params, eval_metric='mlogloss')
```